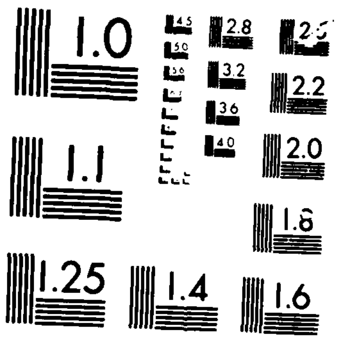
AD-A169 256 AN IMPLICIT ENUMERATION ALGORITHM WITH BINARY-VALUED CONSTRAINTS(U) STANFORD UNIV CA SYSTEMS OPTIMIZATION LAB Y WAN MAR 86 SOL-86-5 N00014-85-K-0343 1/2

UNCLASSIFIED F/G 9/2 NL



MICROCOP[illegible] [illegible]HART

(5)

Systems
Optimization
Laboratory

AD-A169 256

# AN IMPLICIT ENUMERATION ALGORITHM WITH BINARY-VALUED CONSTRAINTS

by

Yieh-Hwang Wan

TECHNICAL REPORT SOL 86-5

March 1986

Department of Operations Research
Stanford University
Stanford, CA 94305

86 7 2 030

(5)

SYSTEMS OPTIMIZATION LABORATORY
DEPARTMENT OF OPERATIONS RESEARCH
STANFORD UNIVERSITY
STANFORD, CALIFORNIA 94305

# AN IMPLICIT ENUMERATION ALGORITHM WITH BINARY-VALUED CONSTRAINTS

by

Yieh-Hwang Wan

TECHNICAL REPORT SOL 86-5

March 1986

Research and reproduction of this report were partially supported by the Office of Naval Research Contract N00014-85-K-0343.

## TABLE OF CONTENTS

| CHAPTER | | PAGE |
|---|---|---|
| 1 | INTRODUCTION . . . . . . . . . . . . . . . | 1 |
| | 1.1. Binary Integer Programming Problem . . | 1 |
| | 1.2. Solution Approaches for Integer Linear Programs . . . . . . . . . . . . . . . . | 3 |
| | 1.3. Model Overview . . . . . . . . . . . . . | 5 |
| 2 | REVIEW OF THE MULTIPLE CHOICE INTEGER PROGRAM . . . . . . . . . . . . . . . . . . | 8 |
| | 2.1. Problem Formulation . . . . . . . . . . . | 8 |
| | 2.2. Branch and Bound Algorithm . . . . . . | 11 |
| | 2.3. Algorithms for MCIP and Related Models . | 14 |
| | 2.4. Branching Schemes . . . . . . . . . . . | 15 |
| 3 | GROUP CONTINGENT CONSTRAINTS . . . | 21 |
| | 3.1. Definitions . . . . . . . . . . . . . . . | 21 |
| | 3.2. Properties of Group Contingent Constraints | 24 |
| | 3.3. Examples . . . . . . . . . . . . . . . . | 33 |
| 4 | STATEMENT OF THE ALGORITHM . . . . . | 38 |
| | 4.1. Model Description . . . . . . . . . . . | 38 |
| | 4.2. The Framework of the Algorithm . . . . | 40 |
| | 4.3. Penalty Computations and Branching Schemes . . . . . . . . . . . . . . . . . | 42 |
| | 4.4. Fathoming Tests . . . . . . . . . . . . | 49 |

4.5. Handling Binary Data . . . . . . . . . . 52
4.5. Examples . . . . . . . . . . . . . . . . . 54
5 HEURISTIC ALGORITHMS . . . . . . . . . . 58
5.1. Introduction . . . . . . . . . . . . . . . 58
5.2. General–Purpose Heuristic Procedures . . 60
5.3. Special–Purpose Heuristic Procedure . . . 63
6 COMPUTATIONAL RESULTS . . . . . . . . 71
6.1. Testing Procedure . . . . . . . . . . . . 71
6.2. Testing Results . . . . . . . . . . . . . 73
6.3. Effects on Different Branching Criteria . . 85
6.4. Testing Results on Procedures H1 and H2 . 88
6.5. Testing Results on Procedure H3 . . . . . 91
7 CONCLUSIONS . . . . . . . . . . . . . . . . 94
7.1. Summary . . . . . . . . . . . . . . . . . 94
7.2. Future Study . . . . . . . . . . . . . . . 95
APPENDIX . . . . . . . . . . . . . . . . . . 97
BIBLIOGRAPHY . . . . . . . . . . . . . . . 101

# CHAPTER 1

# INTRODUCTION

*A common practice for solving a decision making problem is to formulate the problem to fit into a mathematical model whose behavior is well understood. In most cases, the model represents a simplified version of the problem since it is very difficult, if not impossible, to accommodate all the requirements into a comprehensible model. It is expected, however, that the model captures the essence of the problem and provides valuable information to the decision makers. One of the most powerful and widely accepted models is the linear programming (LP) model. The model assumes that all the decision variables are continuous and that all the requirements can be expressed by linear constraints. The objective is to maximize or minimize a linear function of the decision variables subject to the linear constraints. The popularity of the LP model comes from its capability of handling large numbers of variables and constraints, plus other factors such as the ease of conducting postoptimality analysis.*

## §1.1 Binary Integer Programming Problem

The LP model, however, has difficulty in handling discrete decisions. For example, the Lorie–Savage problem [Lorie and Savage, 1955] involves decisions on whether to undertake indivisible projects or not. The decisions are *yes* (undertake) or *no* (reject), which can only be represented by discrete variables. This prompted the development of the integer programming (IP) model, which resembles the LP model in every aspect except for the addition of the integrality requirement. In order to deal with *yes-or-no* decisions, the discrete decision variables need to be further restricted to be binary(0 or 1). In this dissertation, we will present a binary integer programming (BIP) model with certain types of special constraints which arise naturally in the context of capital budgeting, as well as in many other applications, and suggest an algorithm which seems promising in solving the problem more efficiently.

For a mathematical programming problem, the nature of its decision variables usually determines the approach we use to solve the problem. If all the variables are continuous, for example, we might adopt the simplex algorithm (for the LP model) or some nonlinear programming algorithm, depending upon whether nonlinearity exists in the problem, to find an optimal solution. On the other hand, if all the variables are required to be integers, we could use branch and bound algorithms to search for an optimal solution. As reviewed in Section 2.2, there has been substantial progress in the past twenty-five years in developing such algorithms for integer programming. This work has revealed that it usually is advantageous to employ special-purpose algorithms to attack problems with special structure.

Many real-world problems can be formulated naturally as a pure BIP model in which all the decision variables are restricted to the values 0 or 1. These problems are not easy to solve in general. Theoretically, for a pure BIP problem, we can always enumerate all the feasible combinations since they are only of finite number. The difficulty is that the number of feasible combinations could be astronomically large, which prevents us from solving the problem within a reasonable time. An efficient implicit enumeration BIP algorithm should have ways to quickly consider large numbers of potential solutions simultaneously and then to quickly check the optimality of a trial solution. For a structured problem, this sometimes can be achieved by exploiting the special properties it possesses. The classical linear assignment problem is an example, since its special properties guarantee that an optimal solution of the LP-relaxation (obtained by replacing the integrality constraints of the binary variables by constraints stating that the variables can take values between 0 and 1) will be an optimal solution of the original problem, so we can actually use the simplex algorithm to solve the problem. As discussed briefly in the next section, other methods for solving integer programming problems include cutting plane algorithms and the group theoretic approach. It is possible that such methods could be very efficient for certain types of BIP problems. In this dissertation, however, we only consider implicit enumeration algorithms.

## §1.2 Solution Approaches for Integer Linear Programs

There are several different approaches that have been developed to solve integer programming problems. The major ones are enumeration methods, cutting plane methods, and group theory methods. Enumeration methods are by far the most popular approach in practice, and widespread commercial programs now are available. However, other solution approaches are still valuable in the sense that some may be particularly suitable for certain class of problems [Balinski and Quandt, 1964, and Toregas *et al.*, 1971] and some can be used in conjuction with direct enumeration methods to accelerate the enumeration process [Lemke and Speilberg, 1967]. In this section, we will provide a brief description of the cutting plane approach since the concept is used in the proposed algorithm. We will not describe group theory methods here, but do provide a comprehensive list of references. The framework of enumeration methods will be given in the next chapter.

The general principle of the cutting plane approach is to generate linear constraints systematically that are implied by combining the original linear constraints and the integrality requirement on the variables. Cutting plane algorithms can be characterized by the following steps:

(C1) Solve the LP-relaxation of the original problem.

(C2) If there is no feasible solution, stop since the original problem is infeasible.

(C3) If the optimal solution, $x^*$, is a feasible solution for the original problem, stop since it must also be an optimal solution for the original problem.

(C4) Introduce new linear constraints and optimize the LP-relaxation of the revised problem. Let the new optimal solution, if any, be $x^*$. Go to (C2).

The new constraints introduced in (C4) should force the current optimal solution for the LP-relaxation, $x^*$, to become infeasible without eliminating any feasible solutions for the original problem. The key issues for the cutting plane approach include finite termination and the speed of convergence of the algorithm. The idea of introducing implied linear constraints was first used by Dantzig, Fulkerson, and Johnson (1954) and Markowitz and Manne (1957). The cutting plane approach was systematized by Gomory (1958, 1960, 1963). More recent work can be found in, e.g., Balas and Jeroslow (1975), Jeroslow (1979), and Wolsey (1979).

The group theoretic approach was first proposed by Gomory (1965). Subsequent work can be found in Shapiro (1968a, 1968b, 1970), Wolsey (1969), Glover (1969), Gorry and Shapiro (1971), and Johnson (1979,1981). Computational experience on the approach is limited.

Special-purpose algorithms are abundant in solving IP problems, for example, see Reardon (1974) and Kochman (1976). This is appropriate since there is no proven IP algorithm which can solve all IP problems nearly as efficiently as the simplex algorithm can solve LP problems. Furthermore, even the simplex algorithm has difficulty in solving large scale LP problems and needs to take advantage of the special structure.

Good sources for further information on all the topics discussed in this section are Balinski (1965), Beale (1965, 1979), Garfinkel and Nemhauser (1972), Garfinkel (1979), Geoffrion and Marsten (1972), and Geoffrion (1976).

## §1.3 Model Overview

It is not common for a BIP problem to possess the property of the classical linear assignment problem that solving it is equivalent to solving its LP-relaxation. However, many BIP problems do possess other useful special properties. In the context of capital budgeting, mutually exclusive projects and contingent projects [Weingartner, 1963, 1966] often result in constraints with special characteristics which can be used to accelerate the enumeration process. There are two common types of decisions in capital budgeting problems. One is to determine the level of a project to be undertaken, which can be expressed as a nonnegative continuous decision variable, and the other is to decide whether to accept or reject a specific project, which can be expressed as a binary decision variable. In the most general formulation of the capital budgeting problem, both types of variables would appear and the problem is referred to as a mixed integer programming (MIP) problem. However, the focal point of this dissertation is on the mutually exclusive and contingent projects which are associated with the latter category of decisions.

A set of $n$ projects is said to be *mutually exclusive* if we can undertake exactly one project out of these $n$ projects. For example, this may represent the case where we have $n$ investment opportunities and need to select one, or where a computer processor needs to select one of the $n$ waiting tasks to execute next. Mathematically, let

$$x_i = \begin{cases} 1, & \text{if undertake project } i, \\ 0, & \text{if don't.} \end{cases}$$

Then the requirement that the projects be mutually exclusive can be expressed by the constraint

$$\sum_{i=1}^{n} x_i = 1. \tag{1}$$

We call (1) a *multiple choice* constraint. *Contingent projects* occur when there exist precedence relationships among various projects. For example, project 2 is said to be contingent on project 1 if undertaking project 2 requires also undertaking project 1. This can be expressed by the inequality

$$x_1 - x_2 \geq 0. \tag{2}$$

Again, $x_1$ and $x_2$ are binary variables and (2) is called a *contingent* constraint. Note that $x_2 = 1$ implies $x_1 = 1$. Examples of contingent decisions include (a) projects with two stage decisions in which whether to undertake the project at the second stage is contingent upon undertaking its first stage counterpart, and (b) the selecti n of interrelated investment projects where the adoption of certain projects may be conditioned on the outcomes of other projects. A more general definition of contingent constraints will be given formally in Chapter 3. We also call (2) a binary-valued constraint (BVC) since, for every solution satisfying (2), the left-hand-side of (2) can only take the values 0 or 1.

The BIP problem with multiple choice constraints is termed the multiple choice integer program (MCIP). Its formulation and solution techniques will be reviewed in Chapter 2. We then introduce contingent constraints and binary-valued constraints in Chapter 3 and show that many contingent constraints are actually binary-valued.

Several examples of applications involving group contingent constraints also are presented in Chapter 3. The multiple choice constraints and the binary-valued constraints have a tendency to appear simultaneously. Adding the BVC into the basic MCIP model, the resulting model is called MCIP/BVC, which will be the focal point of this dissertation. The proposed algorithm and its applications are given in Chapter 4. In Chapter 5, we introduce three heuristic versions of the proposed algorithm. These heuristic algorithms are designed to find a satisfactory (but not necessarily optimal) solution with considerably less computer time. Numerical experiments and implications are reported in Chapter 6 to demonstrate the efficiency of the new algorithm along with its heuristic counterparts. We conclude this dissertation by summarizing the early results and their implications, and pointing out potential areas for future research in the last chapter.

# CHAPTER 2

# REVIEW OF THE MULTIPLE CHOICE INTEGER PROGRAM

*Before we discuss the proposed algorithm for the multiple choice integer program with binary-valued constraints, it would be beneficial to know the alternatives we have for solving MCIP problems, since MCIP/BVC is a special case of MCIP. In the following sections, we first present the MCIP formulation and then introduce algorithms which are available for solving MCIP type problems. We will highlight various branching schemes that are commonly adopted in these algorithms. The chapter concludes with a comparison of the different branching schemes.*

## §2.1 Problem Formulation

For a general MCIP, we have

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} & \\ \text{subject to} & Ax \geq b & \\ & \sum_{j=1}^{n_i} x_{ij} = 1, \qquad i = 1, 2, \ldots, m & \\ & x_{ij} \in \{0, 1\}, \ \forall \ i, j. & \end{array} \qquad (1)$$

We call each set of variables $\{x_{i1}, x_{i2}, \cdots, x_{in_i}\}$ a special ordered set (SOS)*. We further assume that each variable belongs to exactly one SOS. Hence, the problem is to select one variable to have the value 1 out of each SOS and then set all other variables equal to 0.

Following are several possible extensions of the above formulation:

* Beale and Tomlin (1969) call this a special ordered set of type 1.

(a) If instead of selecting *exactly* one variable out of each SOS, we are allowed to pick *at most* one variable from each SOS, then the multiple choice constraint becomes

$$\sum_{j=1}^{n_i} x_{ij} \leq 1,$$

which we term a generalized upper bounding (GUB) constraint. In this case, we can add a binary slack variable to this constraint and then set the coefficient of this variable equal to 0 in the objective function. This will lead to our earlier formulation.

(b) If there are overlapping variables among different special ordered sets, we can introduce dummy variables into the system and convert it into the formulation (1). For example, suppose that for some $i, j,\ i \neq j$, we have

$$SOS_i \bigcap SOS_j = \{x_k\} \neq \emptyset,$$

where $SOS_i$ denotes the $i$th SOS. Then we can create a dummy variable $x'_k$ and define a new special ordered set $j'$ where

$$SOS_{j'} = SOS_j - \{x_k\} + \{x'_k\}.$$

Replace the $j$th SOS by the $j'$th SOS and set the cost and general constraint coefficients of $x'_k$ equal to 0 in the objective function and all the other constraints. Furthermore, we add a side constraint $x_k = x'_k$, which can be accommodated as a general constraint ($Ax \geq b$). The new formulation is equivalent to the old one but with fewer overlapping variables. This procedure can be applied repeatedly until all overlapping variables are eliminated.

(c) If there are some binary variables, say $x_k$, that do not belong to any SOS, one can introduce a (redundant) constraint,

$$x_k \leq 1,$$

for each such variable and then proceed as described for (a) above to reach the canonical formulation.

(d) It is also possible to consider constraints of the form

$$\sum_{j=1}^{n_i} x_{ij} = k, \tag{2}$$

where $k$ is an integer greater than or equal to 2. One way to get around this problem is to form $k$ copies of the original variables. For example, if

$$\sum_{j=1}^{n_i} x_{ij} = 2, \tag{3}$$

we introduce two sets of binary variables, $\{x_{i1}^1, x_{i2}^1, ..., x_{in_i}^1\}$ and $\{x_{i1}^2, x_{i2}^2, ..., x_{in_i}^2\}$, and constraints $x_{ij}^1 + x_{ij}^2 \leq 1$ for each $j = 1, 2, ..., n_i$. Replace variables $x_{ij}$ by $(x_{ij}^1 + x_{ij}^2)$ in the original formulation for each $j$ and eliminate constraint (3). Then the modified system plus the constraints

$$\sum_{j=1}^{n_i} x_{ij}^1 = 1, \qquad \sum_{j=1}^{n_i} x_{ij}^2 = 1,$$

and the $n_i$ constraints stated above constitute a new system which is equivalent to the original system. This procedure also can be applied to other constraints of the form (2) when $k > 2$. However, this approach is cumbersome since originally for each such constraint we have $\binom{n_i}{k}$ feasible combinations, where $\binom{n_i}{k}$ is the usual combinatorial notation defined as

$$\binom{n_i}{k} = \frac{n_i!}{k!(n_i - k)!}.$$

In the latter formulation we have $k!\binom{n_i}{k}$ combinations, which is far larger.

## §2.2 Branch and Bound Algorithms

*Branch* and *bound* is an optimization technique that utilizes a tree structure to enumerate potential solutions. It involves selecting promising problems to investigate in the enumeration tree and calculating bounds on the objective function. It is basically a strategy of *divide* and *conquer*. The expression *branch* and *bound* was first used in Little *et al.* (1963). During the past twenty-five years, we have witnessed substantial progress in developing such algorithms. Subsequent treatments of the approach can be found in Lawler and Wood (1966), Mitten (1970), and Garfinkel (1979). We will give a brief description of the elements of branch and bound algorithms and then introduce relevant terminologies which are used later in this dissertation.

For definiteness, we assume that the objective function is to be minimized. Suppose that at the start of the algorithm there exists a known feasible solution to the problem with objective value $Z_u$ (if none exists, $Z_u = \infty$). This solution is called an *incumbent*. The algorithm begins by separating the feasible region of the original problem into several (disjoint, if possible) subregions which are defined by imposing constraints in addition to the original constraints. We refer to this procedure as *separation* or *partition*. Each subregion corresponds to a *node* in the enumeration tree, and defines a new optimization problem that is a reduced version of the original problem. These reduced problems are called *subproblems* or *descendants*. The edge in the tree that connects the current node and a new node is called a *branch* which imposes new constraints. At each separation, some variables might become fixed and others called *free* variables are still unrestricted. For each node, we try either to find better feasible solutions to the original problem or to verify that no such solutions exist to warrant further partitioning of the new problem. If successful in the former case, we will update the incumbent by replacing it with the best newly found solution. If successful in the latter case, or in the case that the newly found solution is an optimal solution to the new problem, we don't need to further partition the new problem and we say that the node is *fathomed*. When a fathomed node is detected, we *backtrack* in the enumeration tree until an unfathomed node is found, at which point the procedure repeats itself. The procedure terminates when all nodes become fathomed. We then have an optimal solution ($Z_u < \infty$) or the problem is infeasible ($Z_u = \infty$). Verifying that no better feasible solutions exist involves evaluating bounds, as well as other logic tests which may depend upon the characteristics of the problems. Many fathoming devices and measures which aid the decision on branching have been developed. We will mention some of them briefly.

The use of surrogate constraints was first introduced by Glover (1965). They are basically nonnegative weighted linear combinations of the original constraints and the objective function, and often serve as a fathoming device. Depending upon specific needs, different approaches exist to generate surrogate constraints. Glover (1968) presented many definitions that measure the strength of surrogate constraints. Geoffrion (1969) introduced a surrogate constraint which is designed for the BIP problem and the computation results were very encouraging. Surrogate constraints have been used for other purposes; for example, see Chapter 5.

Penalties are widely adopted as a fathoming tool in branch and bound type algorithms. They can also be used as a criterion for choosing potential branches for further investigation. We will discuss penalties in detail in Chapter 4.

Pseudocosts, which were originated by Benichou *et al.* (1971), serve as a guide for the selection of future branches. They are designed to estimate the change in objective value given that some variables are forced to take certain values. Pseudo-costs can not be used as a fathoming tool since they are only estimates rather than actual bounds. Successful use has been reported by Gauthier and Ribière (1977).

There are many tradeoffs that need to be considered in any branch and bound algorithm. For example, how many subregions should be divided at each partition and should the subregions be roughly the same size? Another tradeoff is between the computational effort devoted to obtaining bounds and the tightness of these bounds. In addition, when we partition in order to introduce new subproblems (new branches leading to new nodes) into the enumeration tree, we need to decide which problem to investigate first. A similar question arises when we backtrack in the enumeration tree and need to decide which unfathomed node to examine next. These decisions are not easy to make and may have great impact on the efficiency of the algorithm. In fact, the behavior of enumeration type algorithms is difficult to predict in general. They might perform very differently on various sets of seemingly similar problems. Usually the only way to judge the comparative merits of such algorithms is to conduct extensive numerical experiments, and even that may not be enough to draw statistically significant conclusions.

Implicit enumeration is the name of a class of branch and bound algorithms for the case in which all variables are required to be binary. This case merits special attention because many real-world decision variables are naturally binary. Having binary variables also enables the derivation of efficient fathoming tests for the algorithm. The algorithm proposed in this dissertation falls into the category of implicit enumeration. The specifics of the implicit enumeration algorithm pertaining to the proposed algorithm will be addressed in Chapter 4. For further discussion of implicit enumeration and related algorithms, see Balas (1965), and Geoffrion and

Marsten (1972).

## §2.3 Algorithms for MCIP and Related Models

Multiple choice type constraints were first analyzed by W. C. Healy, Jr. in his 1964 paper [Healy, 1964]. The problem he proposed is actually an extension of the MCIP model. It allows continuous variables as well as multiple choice variables. Unfortunately, the solution procedure he suggested is not guaranteed to find an optimal solution, or even a feasible one.

The multiple choice knapsack problem, which is a special case of the MCIP problem, has been studied extensively. The problem takes the form of the MCIP model but with only one general constraint. Because of its special structure, special algorithms are designed to accelerate the solution process. For example, the algorithm by Sinha and Zoltners (1979a) begins by investigating the relationships between the coefficients $\{c_{ij}\}$ and $\{a_{ij}\}$. Some variables may be eliminated from consideration if certain conditions are met. It then adopts a streamlined branch and bound algorithm to solve the (possibly) reduced problem. Other specialized algorithms also exist; for example, see Nauss (1975).

The multiple choice integer programming problem is also referred to as the multi-item scheduling problem by Lasdon and Terjung (1971). Several algorithms have been developed to solve MCIP. Sinha and Zoltners (1979b) extended their work on the multiple choice knapsack problem to MCIP. Because of the efficiency of their algorithm for the former problem, they derived a way to combine general constraints into a single surrogate constraint and then applied this algorithm. Another algorithm was presented by Mevert and Suhl (1977). They use a branch and bound approach to solve the problem. Bean (1980) described an additive algorithm for the problem. It is basically a branch and bound approach. Unlike many other algorithms, the enumeration tree implicitly considered in Bean's algorithm is usually not binary. This algorithm utilizes the mutually exclusive property among the variables in a SOS (to be discussed in Section 2.4) to form an enumeration tree and often there are many descendants for each node which are not yet fathomed. Detailed branching schemes are presented in the next section.

## §2.4 Branching Schemes

We mentioned in Section 2.2 that the method of branching (defining new subregions), is one of the key issues in designing enumeration type algorithms. In the context of implicit enumeration, there are basically three types of branching schemes for MCIP. We can illustrate these branching schemes through the following example. Suppose that we have a problem with 6 binary variables and the constraints

$$x_1 + x_2 + x_3 = 1, \tag{SOS 1}$$

$$x_4 + x_5 + x_6 = 1. \tag{SOS 2}$$

According to the usual branch and bound method (B1), we can partition on a selected variable, thereby introducing two new branches and two new nodes corresponding to the fixed values of zero and one for this variable. The partitioning procedure repeats at each node that is not fathomed. In actuality, many nodes may be fathomed very early in the process. However, the *maximum* number of branches that needs to be enumerated explicitly in order to solve the problem is $\sum_{k=1}^{6} 2^k = 126$. We are interested in this quantity since whenever a new branch is created, a revised problem needs to be solved, and the more potential problems we have, the longer it is likely to take to find an optimal solution. However, we must emphasize that this quantity is a worst-case estimate that greatly exceeds the actual number of branches that normally would need to be enumerated, so we use it only as a rough measure to compare various branching schemes.

For the constraints (SOS1) and (SOS2), we note that only three combinations satisfy (SOS1), namely,

$$x_1 = 1, \quad x_2 = 0, \quad x_3 = 0;$$

$$x_1 = 0, \quad x_2 = 1, \quad x_3 = 0;$$

$$x_1 = 0, \quad x_2 = 0, \quad x_3 = 1;$$

similarly, for (SOS2), we have

$$x_4 = 1, \quad x_5 = 0, \quad x_6 = 0;$$

$$x_4 = 0, \quad x_5 = 1, \quad x_6 = 0;$$

$$x_4 = 0, \quad x_5 = 0, \quad x_6 = 1.$$

Thus, there are only $3^2 = 9$ combinations that need to be examined and the maximum number of potential branches is $\sum_{i=1}^{2} 3^i = 12$. This observation leads to the algorithm proposed by Bean (1980), and we call this branching scheme 2 (B2). In general, the maximum number of potential branches equals $\sum_{i=1}^{m} n_0^i$†, assuming that there are $m$ SOS constraints with $n_0$ variables in each SOS.

The third kind of branching rule,(B3), is the dichotomy proposed by Beale and Tomlin (1969). Observe that, for (SOS1), we can either have

$$x_1 + x_2 = 1, \tag{3}$$

† If the number of variables differs among SOS's, then the expression will be more complicated since the order we introduce the SOS constraints into the enumeration tree becomes relevant.

or

$$x_3 = 1. \tag{4}$$

Equation (3) can be further partitioned into $\{x_1 = 1 \text{ or } x_2 = 1\}$. There are other ways to partition the (SOS1), for example, we can have $\{x_1 + x_3 = 1 \text{ or } x_2 = 1\}$ instead of constraints (3) and (4), and then partition the constraint $x_1 + x_3 = 1$ in an obvious way. For a general multiple choice constraint

$$\sum_{i \in N} x_i = 1, \quad N \text{ contains at least two elements,}$$

the branching scheme works as follows:

(D1) Select non-empty disjoint proper subsets $N_1$ and $N_2$ of $N$ such that $N_1 \bigcup N_2 = N$.

(D2) Two branches, respectively associated with the constraints $\sum_{i \in N_1} x_i = 1$ and $\sum_{i \in N_2} x_i = 1$, are created. Repeat step (D1) while replacing $N$ by $N_1(N_2)$, if $N_1(N_2)$ contains more than one element.

(D3) If $N_1(N_2)$ contains a single element, no further partition is necessary for this branch.

One can prove that no matter what order the partition is made, the maximum number of possible branches remains a constant. We have the following proposition:

**Proposition 2.4.1.** *Given the multiple choice constraint*

$$\sum_{i=1}^{n} x_i = 1 \tag{5}$$

*and the branching scheme (B3), then the maximum number of potential branches that can be generated is 2(n-1).*

*Proof.* We prove this proposition by induction.

When $n = 1$, the maximum number of potential branches equals 0 since no branch is needed.

Assume that the proposition is true for all $n$, $n < m$.

When $n = m$, we can first partition the constraint into 2 branches and have $n_1$ and $n_2$ variables in the respective branches, where $n_1 + n_2 = n$. Since $n_1 > 0$ and $n_2 > 0$, this implies that $n_1 < m$ and $n_2 < m$ and the induction hypotheses can be applied. The maximum number of potential branches for the $n_1$ and $n_2$ branches are $2(n_1 - 1)$ and $2(n_2 - 1)$, respectively. Therefore for (5), the maximum number of potential branches is

$$2(n_1 - 1) + 2(n_2 - 1) = 2(n - 2),$$

which is independent of $n_1$ and $n_2$. Counting the initial partition of 2 branches, the *maximum number of potential branches for the original constraint is*

$$2(n - 2) + 2 = 2(n - 1),$$

which concludes the proof. ▮

In the present case, we have $2(3 - 1) = 4$ branches for each SOS. We need examine $4^2 = 16$ branches at the most to solve the problem.

Table 2–1 : Comparison of Branching Schemes

| Branching Scheme | Number of Potential Branches |
|---|---|
| B1 | $\sum_{k=1}^{n} 2^k$ |
| B2 | $\sum_{i=1}^{m} n_0^i$ |
| B3 | $\sum_{i=1}^{m} 2^i(n_0 - 1)^i$ |

In general, for an MCIP with $m$ special ordered sets and $n_0 \geq 2$ variables in each SOS ‡, there are $n = mn_0$ variables in total. The number of potential branches for branching schemes are shown in Table 2-1. For example, for a problem with $m = 5$, $n_0 = 10$, we will get $\approx 2.25 \times 10^{15}, 1.11 \times 10^5$, and $\approx 2.0 \times 10^6$ possible branches, respectively. One can see that (B2) or (B3) require enumerating far fewer potential branches than (B1). Also note that by using (B2) or (B3) we don't need to check the feasibility of any potential solutions with respect to the corresponding multiple choice constraints. Consequently, we have fewer constraints that need to be considered explicitly. These ideas prompt the development of a special-purpose algorithm for solving MCIP/BVC problems.

Branching schemes (B2) and (B3) illustrate the difficulties involved in comparing algorithms. As indicated earlier, the *maximum* number of potential branches is only one measure for comparing branching schemes. In the example above, (B2) generates fewer branches than (B3). However, (B3) has the advantage that at each separation we have more flexibility in assigning values to free variables and we have a more powerful fathoming device. For example, suppose that the problem corresponding to the current node is (P) and the branch to investigate next, according to (B3), is represented by the constraint

$$x_1 + x_2 + \cdots + x_k = 1.$$

Also suppose that the new node is fathomed by some test. In order to achieve the same result by using (B2), we have to examine $k$ new nodes where the problem defined at each node is (P) plus a constraint $x_i = 1$, for some $i = 1, 2, \cdots, k$. This suggests that more than one factor needs to be taken into consideration when we se-

‡ If $n_0$ equals 1, the case is trivial and of little interest to us.

lect the branching scheme for a particular algorithm. The Beale–Tomlin dichotomy (B3) will be the principle branching tool, for reasons to become apparent later, in the MCIP/BVC algorithm.

# CHAPTER 3

# GROUP CONTINGENT CONSTRAINTS

*In this chapter, we will formally introduce group contingent constraints and discuss some of their properties. We show how we can take advantage of this special structure to accelerate the enumeration process. In the last section, we provide examples containing group contingent constraints.*

## §3.1 Definitions

Recall from Section 1.3 that a constraint $f$ of the form

$$\begin{aligned} &f(x,y) = x - y \geq 0, \\ &\textit{where} \quad x \in \{0,1\}, y \in \{0,1\}, \end{aligned} \tag{1}$$

is a contingent constraint. $f(x,y)$ is a binary-valued function in the sense that $f(x_0,y_0) = 0$ or $1$ for every feasible solution.

**Definition 3.1.1.** *We call a constraint $f(\mathrm{x}) \geq 0$ binary–valued if all the variables are binary and $f(\mathrm{x})$ can only take the values 0 or 1 for any feasible combination with respect to the constraint.*

The notion of contingency can be generalized to contain variables from special ordered sets. Letting $Z_j = \{1, 2, \ldots, j\}$, $j \geq 1$, we have the following definition for the *group contingent constraint* (GCC).

**Definition 3.1.2.** *Suppose that we have $m$ mutually exclusive special ordered sets with variables $x_{i1}, x_{i2}, \ldots, x_{in_i}$ in the $i$th set. The group contingent constraint takes the form*

$$\sum_{i \in I_1} \sum_{j \in N_i} x_{ij} - \sum_{i \in I_2} \sum_{j \in N_i} x_{ij} \geq 0$$

$$\text{where} \quad I_1 \neq \emptyset,\ I_1 \subset Z_m,\ I_2 \neq \emptyset,\ I_2 \subset Z_m,\ I_1 \bigcap I_2 = \emptyset, \tag{2}$$

$$N_i \subset Z_{n_i},\ \text{and}\ \ N_i \neq \emptyset \quad \forall\ i \in I_1 \bigcup I_2.$$

In definition 3.1.2, we have assumed that $I_1$ and $I_2$ are disjoint index sets. This assumption is only technical and can be easily relaxed. By not requiring $I_1$ and $I_2$ to be disjoint, we can actually slightly strengthen the fathoming devices stated in the proposed algorithm. In addition, when we refer to a constraint, $f(\mathbf{x}) \geq 0$, as a group contingent constraint in later sections, we mean the constraint $f(\mathbf{x}) \geq 0$ with the understanding that each variable belongs to a special ordered set.

**Example:** **(group contingent constraint).**

*Let* $m = 4,\ n_1 = n_2 = n_3 = n_4 = 3,$

$I_1 = \{1\},\ I_2 = \{3, 4\},$

$N_1 = \{1, 2\} \subset Z_3,\ N_3 = \{2, 3\} \subset Z_3,\ N_4 = \{1\} \subset Z_3,$

then the group contingent constraint defined by (2) is

$x_{11} + x_{12} - x_{32} - x_{33} - x_{41} \geq 0.$

Note that SOS constraints are implicitly considered.

We can characterize a group contingent constraint by the cardinalities of the index sets $I_1$ and $I_2$. Let $|A|$ be the cardinality of the set A. We define

- *group contingent constraint of type SS* ($I_1$ contains $\underline{S}$ingle element and $I_2$ contains $\underline{S}$ingle element) : This is a special case where we have $|I_1| = |I_2| = 1$ in (2). Only two special ordered sets are involved. We also call it a *simple* group contingent constraint. Type SS constraints are binary-valued.

- *group contingent constraint of type SM* ($I_1$ contains $\underline{S}$ingle element and $I_2$ contains $\underline{M}$ultiple elements) : If $|I_1| = 1$ and $|I_2| = m_2 > 1$ in (2), then the resulting constraint (2) is called a group contingent constraint of type SM. Type SM constraints are also binary-valued.

- *group contingent constraint of type MS*: Type MS constraints are obtained by setting $|I_1| = m_1 > 1$ and $|I_2| = 1$ in (2). Contrary to type SM constraints, type MS constraints are not binary-valued.

- A group contingent constraint not of type SS, SM, or MS is called a *general* group contingent constraint.

Variables in a group contingent constraint can be divided into two categories. They are

- *primary variable*: Variables in a group contingent constraint with coefficient 1 are called primary variables with respect to the constraint.

- *secondary variable*: Variables in a group contingent constraint with coefficient $-1$ are called secondary variables with respect to the constraint. We also refer to secondary variables as contingent variables.

In the above example, $x_{11}$ and $x_{22}$ are primary variables, whereas $x_{32}, x_{33}$ and $x_{41}$ are contingent variables. Note that either primary or contingent variables are defined with a specific group contingent constraint in mind. A variable can be a primary variable with respect to one constraint and a contingent variable with respect to another constraint.

We next define the relative strength of the group contingent constraints. This concept can be useful in eliminating redundant constraints.

**Definition 3.1.3.** *A group contingent constraint $f(\mathrm{x}) \geq 0$ is weaker than the group contingent constraint $g(\mathrm{x}) \geq 0$ if the feasible region defined by $g(\mathrm{x})$ is a subset of the feasible region defined by $f(\mathrm{x})$.*

**Example: (relative strength of GCC's).**

Consider the two constraints

$$f(\mathrm{x}) = x_{11} + x_{12} - x_{21} - x_{22} \geq 0 \quad \text{and}$$

$$g(\mathrm{x}) = x_{11} + x_{12} - x_{21} - x_{22} - x_{31} - x_{32} \geq 0.$$

It is clear that the constraint $f(\mathrm{x}) \geq 0$ is weaker since it defines a feasible region larger than that defined by $g(\mathrm{x}) \geq 0$.

## §3.2 Properties of Group Contingent Constraints

Some properties of the group contingent constraints will be explored in this section. To facilitate our discussion, the following conventions are used:

(a) $x_{ij}$'s represent primary variables and $y_{ij}$'s represent contingent variables;

(b) if $|I_1| = 1$, then we write $x_j$ instead of $x_{ij}$ for the primary variables; furthermore, we assume that these $x_{ij}$'s come from the first SOS, i.e., $i = 1$;

(c) if $|I_2| = 1$, then we write $y_j$ instead of $y_{ij}$ for the contingent variables; also we assume that these $y_{ij}$'s are the variables of the last SOS, i.e., $i = m$;

(d) index sets $N_i, i = 1, 2, \ldots, m$, are non-empty proper subsets of $Z_{n_i}$; and

(e) $\overline{N}_i = Z_{n_i} - N_i$ for $i = 1, 2, \ldots, m$.

## §3.2.1 Simple Group Contingent Constraints

Our goal is to utilize the special properties that each type of GCC possesses and derive efficient ways to handle these constraints. For a type SS constraint

$$\sum_{j\in N_1} x_j - \sum_{j\in N_m} y_j \geq 0, \tag{3}$$

$\sum_{j\in N_1} x_j$ can only take the values 0 or 1. When $\sum_{j\in N_1} x_j = 1$, (3) can be eliminated from future consideration, since

$$\sum_{j\in N_1} x_j = 1 \quad \Longrightarrow \quad \sum_{j\in N_m} y_j \leq 1,$$

which always holds because of the constraint $\sum_{j\in Z_{n_m}} y_j = 1$. Consequently, variables in SOS$_1$ which do not belong to N$_1$ can be eliminated and the reduced problem contains fewer constraints and variables. On the other hand, if $\sum_{j\in N_1} x_j = 0$, we need to set variables $y_j, j \in N_m$, equal to 0, since

$$\sum_{j\in N_1} x_j = 0 \quad \Longrightarrow \quad \sum_{j\in N_m} y_j \leq 0,$$

whereas $y_j \geq 0$ for all $j$. This again enables eliminating constraint (3) from future consideration. The procedure we described above, i.e., set $\sum_{j\in N_1} x_j = 1$ or $\sum_{j\in N_1} x_j = 0$ to simplify constraint (3), will be referred to as *partitioning* the constraint in later sections. In addition, we call $\sum_{j\in N_1} x_j \geq 1$ and $\sum_{j\in N_1} x_j \leq 0$ the 1-branch and 0-branch, respectively.

In both cases described above, we need to set certain variables to the value 0. From the computational point of view, there are at least two ways to accomplish this task. In the latter case, for example, we can either use the constraint $\sum_{j\in N_m} y_j \leq 0$ or set their corresponding cost coefficients to a very large constant M (the Big-M method). Advantages of altering cost coefficients are

(a) by not adding new constraints, the dimension of the basis matrix remains the same over the entire solution process, and

(b) ease of implementation.

However, we also know that by using the Big-M method, it usually

(a) requires more pivot steps to reoptimize the problem, and

(b) creates numerical problems.

In our testing code, we adopted the Big-M method mainly for its ease of implementation. Consequently, we have compromised the efficiency of the tested version of the proposed algorithm.

The partitioning procedure has other advantages. Ignore the integral requirement on the variables $x_{ij}$'s for the moment. Observe that when we partition (3) into two alternatives according to whether the value of the first summation is 0 or 1, we have strengthened the constraint implicitly. For example, $x_j$'s and $y_j$'s satisfying

$$\begin{aligned} &\sum_{j \in N_1} x_j = \frac{1}{2}, \sum_{j \in \overline{N}_1} x_j = \frac{1}{2}, \\ &\sum_{j \in N_m} y_j = 0, \sum_{j \in \overline{N}_m} y_j = 1, \end{aligned} \tag{4}$$

are feasible solutions of (3). However, (4) contains no feasible solution for the original problem (with the integral requirement) and the feasible combinations of (4) are also excluded by our partitioning procedure.

## §3.2.2 Type SM Group Contingent Constraints

Type SM constraints are of the form

$$\sum_{j \in N_1} x_j - \sum_{i \in I_2} \sum_{j \in N_i} y_{ij} \geq 0. \tag{5}$$

They are more complex than the simple GCC's. However, the basic approach to partitioning (3) remains valid here. We know that

$$\sum_{j\in N_1} x_j + \sum_{j\in \overline{N}_1} x_j = 1 \quad \Longrightarrow \sum_{j\in N_1} x_j = 1 - \sum_{j\in \overline{N}_1} x_j$$

so (5) becomes

$$1 - \sum_{j\in \overline{N}_1} x_j - \sum_{i\in I_2} \sum_{j\in N_i} y_{ij} \geq 0.$$

Rearranging terms, we have

$$\sum_{j\in \overline{N}_1} x_j + \sum_{i\in I_2} \sum_{j\in N_i} y_{ij} \leq 1. \tag{6}$$

Constraints (5) and (6) are equivalent since they define the same feasible region. In analyzing (6), first partition it on $SOS_1$ (or any other SOS whose index belongs to $I_2$). If $\sum_{j\in \overline{N}_1} x_j = 1$, then

$$\sum_{i\in I_2} \sum_{j\in N_i} y_{ij} \leq 0.$$

Since $y_{ij} \geq 0$ for all $i, j$, we have $y_{ij} = 0$ for all $i \in I_2$, $j \in N_i$. We then apply the procedure as described for the type SS constraints to fix $y_{ij}$'s at value 0 and eliminate (5). For the opposite branch, $\sum_{j\in \overline{N}_1} x_j = 0$, (6) becomes

$$\sum_{i\in I_2} \sum_{j\in N_i} y_{ij} \leq 1, \tag{7}$$

which is still an active constraint and cannot be eliminated. However, constraints (6) and (7) have the same form, so the partition procedure applied to (6) can again be applied to (7). Successively using this partition, the number of free variables will be reduced at each iteration, so (5) will be eliminated eventually. An interesting property of the type SM constraints is that the feasible region they define can be duplicated by a set of simple contingent constraints.

**Proposition 3.2.1.** *The feasible region defined by (5) can be expressed as the finite intersection of feasible regions defined by type SS constraints.*

*Proof.* Let $I = I_2 \bigcup \{1\}$, and define

$$M_i = \begin{cases} \overline{N}_i, & \text{if i=1;} \\ N_i, & \text{if } i \in I_2. \end{cases}$$

Also let

$$\overline{M}_i = Z_{n_i} - M_i, \qquad \text{for} \quad i = 1, 2, \ldots, m.$$

Rewriting (6), we have

$$\sum_{i \in I} \sum_{j \in M_i} y_{ij} \leq 1. \tag{8}$$

Consider the set of constraints

$$\sum_{j \in M_{i_1}} y_{i_1 j} + \sum_{j \in M_{i_2}} y_{i_2 j} \leq 1, \quad \forall\, i_1 \in I, i_2 \in I, i_1 \neq i_2. \tag{9}$$

We show that (8) and (9) are equivalent.

Since $y_{ij} \geq 0$ for all $i,j$, every feasible combination of (8) is feasible in (9). Conversely, for every feasible combination of (9), either

$$\sum_{j \in M_i} y_{ij} = 0, \qquad \forall\, i \in I,$$

which is feasible in (8), or

$$\sum_{j \in M_k} y_{kj} = 1, \qquad \text{for some } k \in I, \tag{10}$$

since all $y_{ij}$'s are integers. (9) and (10) imply that

$$\sum_{j \in M_i} y_{ij} = 0 \qquad \text{for } i \in I, i \neq k. \tag{11}$$

(10) and (11) characterize a feasible combination for (9) which is also feasible in (8).

Rewriting (9), we have

$$\sum_{j \in M_{i_1}} y_{i_1 j} + \left(1 - \sum_{j \in \overline{M}_{i_2}} y_{i_2 j}\right) \leq 1.$$

Rearranging the above expression and multiplying both sides by $-1$, we get

$$\sum_{j \in \overline{M}_{i_2}} y_{i_2 j} - \sum_{j \in M_{i_1}} y_{i_1 j} \geq 0 \quad \forall\, i_1 \in I, i_2 \in I, i_1 \neq i_2,$$

which is a set of type SS constraints. ∎

Proposition (3.2.1) shows that for any type SM constraint, we can transform it into a set of type SS constraints with the identical feasible region. However, the number of type SS constraints required usually is quite large. Using the procedure described in the proof of this proposition, we need $\binom{m_2+1}{2}$ type SS constraints. Generally, this conversion is very cumbersome and is not recommended for real applications. However, it is useful for analyzing type SM constraints.

§3.2.3 Type MS Group Contingent Constraints

Type MS constraints can be treated in a similar way. Let

$$\sum_{i \in I_1} \sum_{j \in N_i} x_{ij} - \sum_{j \in N_m} y_j \geq 0, \quad \text{where } |I_1| = m_1 > 1, \tag{12}$$

be the constraint under consideration. Since

$$\sum_{j \in N_m} y_j + \sum_{j \in \overline{N}_m} y_j = 1 \quad \Longrightarrow \sum_{j \in N_m} y_j = 1 - \sum_{j \in \overline{N}_m} y_j,$$

we have

$$\sum_{i \in I_1} \sum_{j \in N_i} x_{ij} + \sum_{j \in \overline{N}_m} y_j \geq 1. \tag{13}$$

If $\sum_{j \in \overline{N}_m} y_j = 1$, then (13) becomes

$$\sum_{i \in I_1} \sum_{j \in N_i} x_{ij} \geq 0$$

which is always true, so (13) can be eliminated. If $\sum_{j \in \overline{N}_m} y_j = 0$, then (13) becomes

$$\sum_{i \in I_1} \sum_{j \in N_i} x_{ij} \geq 1 \tag{14}$$

which is of the same functional form as (13), so we can repeat the same procedure on (14). Contrary to the type SM constraints, the type MS constraints cannot be represented by the intersection of the type SS constraints.

**Proposition 3.2.2.** *The feasible region defined by (12) cannot be expressed as the finite intersection of feasible regions defined by type SS constraints.*

*Proof.* We first prove this proposition for the case of $|I_1| = 2$. The type MS constraint of (12) has the form

$$\sum_{j \in N_1} x_j + \sum_{j \in N_k} z_j - \sum_{j \in N_m} y_j \geq 0 \tag{15}$$

where the $z_j$'s are the variables from the $k$th SOS. We assume that (15) can be represented by a set of type SS constraints, $W$, and note that we need not consider the variables belonging to the SOS's other than those appearing in (15) because these variables are unconstrained with respect to (15). The feasible region defined by any constraint in $W$ is larger than that defined by (15) since, by assumption, their intersection is the feasible region defined by (15). Let $f(x,y) \geq 0$ be a type SS constraint in $W$ containing variables from $SOS_1$ and $SOS_m$. Then the combinations satisfying

$$\sum_{j \in N_1} x_j = 0, \quad \sum_{j \in N_m} y_j = 1, \quad \sum_{j \in N_k} z_j = 1 \tag{16}$$

are feasible solutions of $f(\mathbf{x},\mathbf{y}) \geq 0$ since they are feasible solutions of (15). This implies that the combinations satisfying

$$\sum_{j\in N_1} x_j = 0, \quad \sum_{j\in N_m} y_j = 1, \quad \sum_{j\in N_k} z_j = 0 \tag{17}$$

are also feasible solutions of $f(\mathbf{x},\mathbf{y}) \geq 0$ because the $z_j$'s do not appear in $f(\mathbf{x},\mathbf{y})$. From (16) and (17), we know that any combination satisfying

$$\sum_{j\in N_1} x_j = 0, \quad \sum_{j\in N_m} y_j = 1$$

is a feasible solution of $f(\mathbf{x},\mathbf{y}) \geq 0$. We already know that the combinations satisfying

$$\sum_{j\in N_1} x_j = 1, \quad \sum_{j\in N_m} y_j = 1;$$

$$\sum_{j\in N_1} x_j = 1, \quad \sum_{j\in N_m} y_j = 0;$$

or

$$\sum_{j\in N_1} x_j = 0, \quad \sum_{j\in N_m} y_j = 0$$

are feasible solutions of $f(\mathbf{x},\mathbf{y}) \geq 0$, since they are feasible combinations of (15). Hence, $f(\mathbf{x},\mathbf{y}) \geq 0$ does not exclude any point with binary coordinates from consideration and is a trivial constraint. A similar argument goes through for any constraint in $W$ of the form $f(\mathbf{x},\mathbf{z}) \geq 0$ as well as $f(\mathbf{y},\mathbf{z}) \geq 0$. Thus all constraints in $W$ are trivial constraints and their intersection is the whole space. This is a contradiction since the combinations satisfying

$$\sum_{j\in N_1} x_j = 0, \quad \sum_{j\in N_m} y_j = 1, \quad \sum_{j\in N_k} z_j = 0$$

are not feasible solutions of (15).

For the case that $|I_1| > 2$, a similar approach can be applied. The only complication is that we need consider more simple GCC's than those mentioned earlier. Hence, we conclude the proof. ∎

## §3.2.4 General Group Contingent Constraints

General group contingent constraints are not binary-valued and cannot be represented by type SS constraints. The focus of this dissertation is on binary-valued constraints, so we are not going to discuss the type MS and general GCC's at great length. How to exploit these special structures is an important topic for future research.

It is not difficult to characterize the weaker GCC's so they can be eliminated from the constraint set. Given a GCC, it becomes stronger when some primary variables are deleted from the constraint or when some contingent variables are added into the constraint. Note that when a type SM GCC is represented by a set of type SS GCC's, the type SM GCC is a stronger constraint than any one of those type SS constraints.

Before we conclude this section, we want to show the relationship between the GCC's and the binary-valued constraints. We call a binary-valued constraint *non-trivial* if there exists at least one variable with a negative coefficient. Assuming that all variables are binary and each variable belongs to exactly one special ordered set, we have the following proposition.

**Proposition 3.2.3.** *A non-trivial binary-valued constraint, with coefficients 1, 0, or -1, is either a simple GCC or a type SM GCC.*

*Proof.* Let $f(\mathbf{z}) \geq 0$ be a non-trivial binary-valued constraint. Rearranging terms, we have

$$f(\mathbf{z}) = f_1(\mathbf{x}) - f_2(\mathbf{y}) \geq 0$$

where x and y consist of variables in z with coefficients 1 and $-1$, respectively. $f_1$ and $f_2$ are defined in an obvious way. Since $f(\mathbf{z}) \geq 0$ is binary-valued, x must consist of variables from only one special ordered set, since otherwise $f(\mathbf{z})$ can take values greater than 1. The non-triviality of $f(\mathbf{z})$ implies that $f_2$ is not identical to 0, so $f(\mathbf{z})$ is either a simple or type $SM$ GCC. ∎

Note that in the above proposition, we dropped the requirement that variables in x and variables in y must come from different special ordered sets (i.e., $I_1 \bigcap I_2 = \emptyset$). Recall that definition (3.1.2) can be relaxed by not requiring $I_1 \bigcap I_2 = \emptyset$ and the proposed solution methodology is still applicable. We will use the terms, binary-valued constraints and group contingent constraints, interchangably in the subsequent presentation with the understanding that the GCC's under consideration are binary-valued.

## §3.3 Examples

Group contingent constraints appear in many applications. Most commonly, they are associated with scheduling type problems. In this section, we give three examples in which binary-valued constraints arise naturally in the formulation. The first example is a generalization of a problem which can be found in Hillier and Lieberman (1980).

### Example 1 : Factory-warehouse problem

Suppose that a company has decided to build a factory in one of $n$ possible locations. In addition, the company is also considering building a warehouse. For each possible factory site, there are several locations where a warehouse can be built. Let

$$x_i = \begin{cases} 1, & \text{if site } i \text{ is selected to build a factory,} \\ 0, & \text{if otherwise,} \end{cases}$$

and

$$y_j = \begin{cases} 1, & \text{if site } j \text{ is selected to build a warehouse,} \\ 0, & \text{if otherwise,} \end{cases}$$

Then the company requirements can be expressed as

$$\sum_{i=1}^{n} x_i = 1,$$

$$\sum_{j=1}^{m} y_j \leq 1,$$

$$x_i - \sum_{j \in N_i} y_j \geq 0, \quad \text{for} \quad i = 1, \ldots, n$$

where all the variables are binary. $m$ represents the number of possible sites for the warehouse and $N_i$ represents the subset of potential sites where a warehouse can be built provided that the factory is located at site $i$. The constraint set consists of multiple choice decisions and contingent constraints, which together represent the basic model which we are going to explore in the next chapter. However, note that in this particular case, the constraint $\sum_{j=1}^{m} y_j \leq 1$ is redundant.

The next example illustrates a scheduling problem where the contingent constraints arise because of geographic requirements. We introduce this example in the context of a road maintenance problem. Interested readers may refer to Armstrong *et al* (1981) for a more detailed description.

**Example 2 :** Road Maintenance Problem

Suppose that there are $n$ road sections that need to undergo maintenance work. For each road section, we have a number of possible maintenance strategies, for example, we may resurface the entire road section or patch the road section as needed. Clearly, these strategies are mutually exclusive. In addition, there are certain geographic constraints. For example, if road sections 1,2 and 3 are in close proximity, we may not want to resurface sections 2 or 3 unless section 1 is to be resurfaced. Requirements of this nature constitute the group contingent constraints.

In mathematical terms, we introduce a binary variable

$$x_{ij} = \begin{cases} 1, & \text{if strategy } j \text{ is adopted for road section } i, \\ 0, & \text{if otherwise,} \end{cases}$$

for all combinations of road sections and strategies under consideration. The multiple choice constraint can be expressed as

$$\sum_{j \in M_i} x_{ij} = 1, \quad \text{for} \quad i = 1, \ldots, n$$

where $M_i$ is the subset of potential road maintenance strategies that is under consideration for road section $i$. To describe the geographic constraints mentioned above, we define index sets $N_{kj}$, where $i \in N_{kj}$ indicates that to adopt strategy $j$ on road section $i$, the $j$th strategy must have been adopted on road section $k$. Contingent constraints are of the form

$$x_{kj} - \sum_{i \in N_{kj}} x_{ij} \geq 0.$$

Other group contingent constraints as well as general constraints may also exist. The resulting problem probably will involve a large number of variables and constraints. Therefore, it may be necessary, or at least advisable, to use heuristic algorithms to solve the problem.

As we have indicated in Chapter 1, many contingent constraints arise from capital budgeting problems. The following example is from Hanssmann(1968). The contingent constraints result from technological dependence among potential projects.

Example 3: Technologically Dependent Projects

Suppose that we have $n$ projects and $T$ time periods. Introduce the binary variables

$$x_{ij} = \begin{cases} 1, & \text{if project } i \text{ selected in period } j, \\ 0, & \text{if project } i \text{ is not selected in period } j, \end{cases}$$

where $i = 1, \ldots, n$ designates the projects and $j = 1, \ldots, T$ designates the point in time when the project is selected. Assuming that each project can be selected at most once, we have

$$\sum_{j=1}^{T} x_{ij} \leq 1,$$

which constitute the multiple choice constraints. Furthermore, if we assume, for example, that project $i$ must be carried out before (or simultaneously with) project $k$, then we have a set of $T$ group contingent constraints

$$\sum_{j=1}^{t} x_{ij} - \sum_{j=1}^{t} x_{kj} \geq 0, \quad t = 1, \ldots, T.$$

Let the budget available in period $j$ be $B_j$ and let $a_{ij}$ be the investment outlay required for project $i$ in period $j$ if it is selected in period $j$. We assume here that each investment would require an outlay only in its initial period and that any unused budget from early periods can not be applied toward future budgetary requirement. Then we have $T$ budgetary constraints, i.e.,

$$\sum_{i=1}^{n} a_{ij} x_{ij} \leq B_j, \quad j = 1, \ldots, T.$$

Similarly, let $E_{ij}$ designate the return of project $i$ if it is selected in period $j$ and $u_j$ designate the return function on the unused budget in period $j$. Then the objective function is to maximize the total return $E$, where

$$E = \sum_{i=1}^{n} \sum_{j=1}^{T} E_{ij} x_{ij} + \sum_{j=1}^{T} u_j (B_j - \sum_{i=1}^{n} a_{ij} x_{ij}).$$

For simplicity, we have ignored cash flows from investment except in their initial period and have not provided a proper discounting factor, but it is not difficult to incorporate such additional factors. If $u_j$'s and any other constraints all turn out to be linear, then this problem fits nicely into the model we are going to discuss in detail in the next chapter. †

The resulting mathematical formulation of Example 3 could become very complicated if the precedence relationships among the projects are complex. Special algorithms may be worth developing for this particular application, but they are beyond the scope of this dissertation.

These examples demonstrate the importance of exploring group contingent constraints.

† Our standard formulation assumes minimization, whereas we maximize the objective function in this example, but converting from maximization to minimization only requires multiplying through the objective function by (-1).

# CHAPTER 4

# STATEMENT OF THE ALGORITHM

*A detailed statement of the model and the algorithm will be presented in this chapter. The specifics of the branching and fathoming schemes are discussed. We also show the data structure of the group contingent constraints. Examples are given in the last section.*

## §4.1 Model Description

We will focus our discussion on the model

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} & \\ \text{subject to} & Ax \geq b & \\ & Gx \geq 0 & (P) \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1, 2, ..., m & \\ & x_{ij} \in \{0,1\} \quad \forall\, i,j & \end{array}$$

where $A$ is a matrix of dimension $l_1 \times \sum_{i=1}^{m} n_i$,

$b$ is a column vector of length $l_1$,

$Gx \geq 0$ represents a set of binary-valued constraints, and

$G$ is a matrix of dimension $l_2 \times \sum_{i=1}^{m} n_i$.

We call (P) a multiple choice integer program with binary-valued constraints (MCIP/BVC). For simplicity, we assume that the BVC's under consideration are simple group contingent constraints. If $l_1 = 1$, we have a knapsack MCIP problem with binary-valued constraints which can be solved by the proposed algorithm very efficiently. If $l_2 = 0$, then (P) is reduced to a MCIP problem.

Problem (P) can be treated as a MCIP problem and solved by the methods described in Chapter 2. Regardless of which method we adopt, there are underlying linear programming problems we have to solve. If the Tomlin-Beale approach is used, the working basis* of the LP problems is of dimension $(l_1 + l_2 + 1) \times (l_1 + l_2 + 1)$. An alternative approach is to consider constraints $Ax \geq b$ and $Gx \geq 0$ independently. We can accomplish this by first excluding $Gx \geq 0$ from our constraint set, which will reduce the size of the working basis to $(l_1 + 1) \times (l_1 + 1)$, and then bring back individual constraints belonging to $Gx \geq 0$. Since we can branch on a binary-valued constraint just as we branch on a binary variable in the usual implicit enumeration algorithms, bringing back such constraints will not alter the dimension of the working basis. Another advantage arises when the model (P) without constraints $Gx \geq 0$, i.e.,

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} & \\ \text{subject to} & Ax \geq b & \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1, 2, ..., m & \qquad (R) \\ & x_{ij} \in \{0, 1\} \quad \forall\, i, j. & \end{array}$$

possesses additional special structure. For example, if

(a) there is only one general constraint, then (R) is a knapsack MCIP problem and efficient algorithms exist to solve (R);

(b) the constraint set of (R) is a totally unimodular matrix, then (R) can be solved by linear programming algorithms; and

(c) (R) is a special network problem, then the procedure to solve the LP-relaxation of (R) can be accelerated.

---

* Working basis is the basis matrix that we need to update at each simplex iteration. See Section 4.3 for further explanation.

These reasons, among others, motivate the algorithm which will be discussed in the following sections.

## §4.2 The Framework of the Algorithm

Before we get into the details of the algorithm, let us first state the framework of the algorithm.

**Algorithm G** (*Solve MCIP/BVC* ).This algorithm is designed to solve MCIP/BVC problems by taking advantage of the fact that certain constraints are binary-valued.

**G0.** [Initialization.] Solve the LP–relaxation of (R). If the problem is not feasible, then (P) is not feasible. Initialize the list map M. Set $Z_{opt} = \infty$.

**G1.** [Integrality?] If there are variables with fractional values, go to G4.

**G2.** [Feasibility?] If the current solution is not feasible in (P), go to G5. This indicates that some binary–valued constraints are violated.

**G3.** [Backtrack.] Update incumbent if necessary. Update list map M. Go to G9.

**G4.** [BVC exhausted?] If all BVC are branched, go to G8.

**G5.** [Branch.] Compute penalties and decide which branch to use next. Update list map M.

**G6.** [Fathom?] Perform various fathoming tests. If fathomed, go to G3.

**G7.** [Reoptimize.] Modify cost coefficients and reoptimize the resulting problem. If there is no feasible solution which has better objective value than the incumbent, go to G3. Otherwise, go to G1.

**G8.** [Solve.] Solve the reduced IP problem by applying an appropriate algorithm for this MCIP problem. Go to G3.

**G9.** [Terminate?] If the list map M is empty, stop. Either the problem is solved or the problem is not feasible. Otherwise, go to G5.

Several points are worth mentioning here. At G0, we initialize the list map M which contains information about how branches have been made, which branches are already fathomed, and which branch should we explore next when we begin to backtrack. $Z_{opt}$ represents the best (smallest) objective value of (P) we have found so far. At G1, we test whether a variable is of integer value or not. Because of the roundoff errors, we consider a positive number $r$ to be an integer if $\min\{f_r, 1 - f_r\} < \epsilon$, where $f_r$ is the fractional part of $r$ and $\epsilon$ is a given small number, usually around $10^{-6}$. At G2, we check the feasibility against the unbranched (yet to be introduced into the reduced system) binary-valued constraints, if any. We backtrack (step G3) in the enumeration tree when a branch is fathomed. At G8, we have an integer programming problem with a smaller number of variables and constraints than the original problem. This problem may possess special structure as discussed in the immediately preceding section. Appropriate algorithm in G8 means an algorithm which suits the special structure of the reduced system. After we solve this problem, this branch is automatically fathomed. The original problem will be declared infeasible if $Z_{opt}$ equals to $\infty$.

Step G5 addresses the question of how to introduce new constraints into the reduced system. Suppose that there still exist unsatisfied binary-valued constraints and we have to decide

(a) whether we should add a new constraint into the system or leave the current branch as is and switch to another unfathomed branch. In the proposed algorithm, we always introduce new constraints into the system.

(b) if adding a new constraint, which constraint should be brought in and which branch (0-branch or 1-branch) should be investigated first.

There is no single best criterion which governs the selection of new branches. In the next section, we introduce the concept of *penalty* associated with each binary-valued constraint. Penalties provide a partial answer to the above questions.

Finally, we note that the algorithm G will terminate in a finite number of iterations. This is clear since

(a) the number of binary-valued constraints to be reintroduced into the system is finite, so the binary tree is of finite length,

(b) the objective value decreases as we update the incumbent and this eliminates the possibility of cycling, i.e., investigating already fathomed branches, and

(c) the appropriate algorithm specified in step G8 should be of finite termination.

## §4.3 Penalty Computations and Branching Schemes

Penalties are very useful in two ways. They can be used to calculate better bounds and hence increase the chance that a branch becomes fathomed earlier. Penalties also serve as an indicator to select promising branches. The use of penalties was proposed by Driebeek (1966). Subsequent work on penalties and related topics can be found in Beale and Tomlin (1969), Tomlin (1970), Davis, Kendrick and Weitzman (1971), to name a few. Initially, penalties are computed with respect to a single integer variable. However, it is not difficult to extend the concept of *penalty* to a set of variables.

We assume that the LP-relaxation of (R) and its succeeding problems † are solved by the generalized upper bounding (GUB) algorithm, and we will discuss penalties in the context of GUB algorithms. For a complete treatment of the GUB algorithm, interested readers are referred to Dantzig and Van Slyke (1967), and Kaul (1965). We will explain several terms that often are used in the algorithm.

† Descendants of (R) with modified cost coefficients.

Recall that we have $l_1$ general constraints and $m$ special ordered sets. Consider the LP-relaxation of (R). It can be easily shown[Dantzig and Van Slyke, 1967] that for any feasible basis for the system, at least one variable from each SOS will appear as a basic variable. Thus, we can choose one basic variable from each SOS and call it the *key* variable. A basic variable which is not a key variable is called a *non-key* variable. Clearly, the selection of key variables is not unique. The *working basis* is composed of columns associated with key variables. We also adopt the following notation:

- $\mathcal{K}$: the set of key variables. (Note that for each SOS, there is only one key variable. This implies that $\mid \mathcal{K} \mid = m$.)
- $\mathcal{B}$: the set of non-key variables. ($\mid \mathcal{B} \mid = l_1 + 1$.)
- $K_i$: the $i$th key variable (assumed to be in the $i$th SOS).
- $B_i$: the $i$th non-key variable.
- $\mathcal{R}$: the set of non-basic variables.
- $S_i$: $S_i = \{ip | p \in Z_{n_i}\}$.
- $T_i$: $T_i = \{ip | p \in N_i\}$.
- $Q_i$: $Q_i = \{t | B_t = i\}$.
- $\overline{T}_i$: $\overline{T}_i = S_i - T_i$.

Suppose that the $q$th binary-valued constraint for which the penalties are calculated is of the form

$$\sum_{j \in T_1} x_j - \sum_{j \in T_2} x_j \geq 0. \tag{1}$$

Two branching alternatives,

$$\sum_{j \in T_1} x_j \geq 1 \tag{2}$$

and

$$\sum_{j\in T_1} x_j \le 0, \qquad (3)$$

are being considered. The task is to choose a constraint for which the branching will take place and decide which branching alternative we should pursue first. Recall that we referred to (2) as a 1–branch and (3) as a 0–branch in Chapter 3.

To illustrate how to compute penalties, consider any optimal solution of the LP–relaxation of the current problem, which is (R) or (R) with certain constraints added, given by

$$\begin{aligned}
x_{B_i} &= \bar{a}_{i00} + \sum_{j\in\mathcal{R}} \bar{a}_{ij}(-x_j), && \text{for } i = 0,1,\ldots,l_1 && (4)\\
x_{K_i} &= 1 - \sum_{\substack{j\in S_i\\ j\neq K_i}} x_j, && \text{for } i = 1,2,\ldots,m && (5)\\
x_j &= 0, && \text{for } j\in\mathcal{R}.
\end{aligned}$$

Observe that in the above formulas, index $j$ has two components and the subscript of $a$ has three components. $x_{00}$ represents the objective value.

For each of the alternatives of (1), a penalty may be computed. Adopting (2) amounts to forcing variables in $\overline{T}_i$ to zero. A pseudo *up penalty* is obtained if a 1–branch is chosen. Specifically, if all basic variables in the first SOS are in $T_1$, then (2) is satisfied and the up penalty equals zero. If the key variable is in $T_1$ but not all non–key variables are in $T_1$, then substituting (5) into (2) yields

$$\sum_{j\in T_1\bigcap\mathcal{R}} x_j + \sum_{j\in T_1\bigcap\mathcal{B}} x_j + \Bigl(1 - \sum_{\substack{j\neq K_1\\ j\in S_1}} x_j\Bigr) \ge 1.$$

Rearranging and cancelling corresponding terms, we get

$$-\sum_{j\in\overline{T}_1\bigcap\mathcal{R}} x_j - \sum_{j\in\overline{T}_1\bigcap\mathcal{B}} x_j \ge 0. \qquad (6)$$

Substituting (4) into (6), this yields

$$-\sum_{j\in\overline{T}_1\bigcap\mathcal{R}} x_j - \sum_{\substack{j\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_j}} \left(\overline{a}_{i00} + \sum_{t\in\mathcal{R}} \overline{a}_{it}(-x_t)\right) \geq 0.$$

Moving constants to the right hand side and exchanging summation signs, we have

$$\sum_{j\in\overline{T}_1\bigcap\mathcal{R}} (-x_j) - \sum_{t\in\mathcal{R}} \sum_{\substack{j\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_j}} \overline{a}_{it}(-x_t) \geq \sum_{\substack{j\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_j}} \overline{a}_{i00}.$$

Finally, combining terms and exchanging indices $t$ and $j$ in the second term, we get

$$\begin{aligned}\sum_{j\in\overline{T}_1\bigcap\mathcal{R}} \left(1 - \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{ij}\right)(-x_j) + \sum_{\substack{j\in\mathcal{R}\\ j\notin\overline{T}_1\bigcap\mathcal{R}}} \left(- \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{ij}\right)(-x_j) \\ \geq \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{i00}.\end{aligned} \tag{7}$$

Denote

$$\overline{a}^*_{qj} = \begin{cases} 1 - \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{ij} & \text{if } j \in \overline{T}_1\bigcap\mathcal{R}; \\ - \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{ij} & \text{if } j \in\mathcal{R},\ j \notin \overline{T}_1\ ; \\ \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B}\\ i\in Q_t}} \overline{a}_{i00} & \text{if } j = 0, \end{cases} \tag{8}$$

and (7) becomes

$$\sum_{j\in\mathcal{R}} \overline{a}^*_{qj}(-x_j) \geq \overline{a}^*_{q0}. \tag{9}$$

If the right hand side of (9) is a positive number, then (9) is not satisfied by the current solution. In other words, it is a valid *cut*. To introduce (9) into the reduced system and find a feasible solution of its LP-relaxation, dual simplex iterations are required. The pivot column will be determined by

$$\frac{\overline{a}_{0k}}{\overline{a}^*_{qk}} = \min\left(\frac{\overline{a}_{0j}}{\overline{a}^*_{qj}}, \overline{a}^*_{qj} < 0\right),$$

and the corresponding increase in $x_{00}$ after the initial dual simplex iteration is $\overline{a}^*_{q0} \cdot \overline{a}_{0k} / \overline{a}^*_{qk}$. Let

$$U_q = \min_{j \in \mathcal{R}} \left( \frac{\overline{a}^*_{q0} \cdot \overline{a}_{0j}}{\overline{a}^*_{qj}}, \overline{a}^*_{qj} < 0 \right), \tag{10}$$

which is an up penalty for partitioning on the $q$th binary–valued constraint. It is a lower bound on the increase in $x_{00}$, since more dual simplex pivots may be required to restore primal feasibility. A similar procedure can be applied to compute an up penalty in the case that the key variable is not in set $T_1$. In particular, we have

$$\overline{a}^*_{qj} = \begin{cases} -1 + \sum_{\substack{t \in T_1 \bigcap \mathcal{B} \\ i \in Q_t}} \overline{a}_{ij} & \text{if } j \in T_1 \bigcap \mathcal{R}; \\ \sum_{\substack{t \in T_1 \bigcap \mathcal{B} \\ i \in Q_t}} \overline{a}_{ij} & \text{if } j \in \mathcal{R},\ j \notin T_1\ ; \\ 1 - \sum_{\substack{t \in T_1 \bigcap \mathcal{B} \\ i \in Q_t}} \overline{a}_{i00} & \text{if } j = 0, \end{cases} \tag{11}$$

and an up penalty can be evaluated accordingly. For the special case where all basic variables are in $\overline{T}_1$, (10) is still valid if the convention $\sum_{j \in \emptyset} z_j = 0$ is adopted. We have

**Proposition 4.3.1.** *An up penalty of (1) can be computed by (10) where the $\overline{a}_{0j}$'s are defined in (4), whereas the $\overline{a}^*_{qj}$'s are defined in (8) if the key variable is in $T_1$ and defined in (11) if the key variable is not in $T_1$.* ∎

For the alternative 0–branch, a pseudo *down penalty* can be computed. We have

$$\sum_{j \in T_1} x_j \leq 0 \qquad \Longrightarrow \sum_{j \in T_2} x_j = 0,$$

so we calculate a down penalty with respect to the constraint

$$\sum_{j \in T_1} x_j + \sum_{j \in T_2} x_j \leq 0.$$

The same procedure as before can be applied to obtain the coefficients $\overline{a}^*_{qj}$'s. However, the results are more complicated because we have to consider whether the two key variables are in $T_1$ and $T_2$, respectively. Cases (a) through (d) summarize the results.

**Case (a).** $K_1 \in T_1$ and $K_2 \in T_2$. Coefficients of (9) are

$$\overline{a}^*_{qj} = \begin{cases} \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} + \sum_{\substack{t\in\overline{T}_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - 1 & \text{if } j \in (\overline{T}_1 \bigcup \overline{T}_2)\bigcap\mathcal{R}; \\ \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} + \sum_{\substack{t\in\overline{T}_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} & \text{if } j \in\mathcal{R},\ j \notin \overline{T}_1 \bigcup \overline{T}_2; \\ 2 - \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} - \sum_{\substack{t\in\overline{T}_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} & \text{if } j = 0. \end{cases}$$

**Case (b).** $K_1 \in T_1$ and $K_2 \notin T_2$. We have

$$\overline{a}^*_{qj} = \begin{cases} \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - 1 & \text{if } j \in \overline{T}_1\bigcap\mathcal{R}; \\ \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} + 1 & \text{if } j \in \overline{T}_2\bigcap\mathcal{R}; \\ \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}ij & \text{if } j \in\mathcal{R},\ j \notin \overline{T}_1 \bigcup \overline{T}_2; \\ 1 - \sum_{\substack{t\in\overline{T}_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} + \sum_{\substack{t\in\overline{T}_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} & \text{if } j = 0. \end{cases}$$

**Case (c).** $K_1 \notin T_1$, and $K_2 \in T_2$.

Results followed by exchanging indices 1 and 2 in case (b).

**Case (d).** $K_1 \notin T_1$, and $K_2 \notin T_2$.

$$\overline{a}^*_{qj} = \begin{cases} 1 - \sum_{\substack{t\in T_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} & \text{if } j \in (T_1 \bigcup T_2)\bigcap\mathcal{R}; \\ -\sum_{\substack{t\in T_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} - \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{ij} & \text{if } j \in\mathcal{R},\ j \notin T_1 \bigcup T_2; \\ \sum_{\substack{t\in T_1\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} + \sum_{\substack{t\in T_2\bigcap\mathcal{B} \\ i\in Q_t}} \overline{a}_{i00} & \text{if } j = 0. \end{cases}$$

A down penalty can be computed as in (10). This gives us

**Proposition 4.3.2.** *A down penalty $D_q$ can be calculated by*

$$D_q = \min_{j \in \mathcal{R}} \left( \frac{\overline{a}^*_{q0} \cdot \overline{a}_{0j}}{\overline{a}^*_{qj}}, \overline{a}^*_{qj} < 0 \right)$$

*where the $\overline{a}_{0j}$'s are defined in (4) and the $\overline{a}_{qj}$'s are defined in cases (a) through (d).*

∎

So far, we only use the fact that an additional constraint needs to be satisfied to compute the penalties. Another property that can be utilized to strengthen the penalty calculations is that the nonbasic variables are required to be integers. In the present case, the nonbasic variables must increase to one and the penalty for such an increase is simply their respective reduced costs. The improved up penalty can be expressed as

$$U^*_q = \min_{j \in \mathcal{R}} \left( \max_{\overline{a}^*_{qj} < 0} \left( \frac{\overline{a}^*_{q0} \cdot \overline{a}_{0j}}{\overline{a}^*_{qj}}, \overline{a}_{0j} \right) \right).$$

An improved down penalty $D^*_q$ can be obtained similarly. Penalties $U^*_q$ and $D^*_q$ are calculated in the algorithm. However, other improvements of the penalty calculation can easily be incorporated into the algorithm.

Penalties can be used as a tool to select the constraint we are going to introduce into the system as well as to select the new branching alternative. As we have mentioned earlier, there is no single best criterion in selecting new branches. Each criterion has its own rationale and merits. One possible strategy is:

(A) Introduce the $k$th GCC where

$$k = \text{Argmax}_q \max \left\{ D^*_q, U^*_q \right\}.$$

(B) If the penalty $P_k = D^*_k$, perform a 1-branch. Otherwise, perform a 0-branch.

We call this the *worst alternative* branching heuristic. The rationale behind this criterion is that the subproblems we store are likely to only contain poor solutions. Hopefully many of them will never be considered after we find some potentially good solutions. Computational results for this criterion and other selection heuristics are available in Chapter 6.

In the case where there are several indices, say $C=\{k_1, k_2, \ldots, k_\alpha\}$, such that $P_k = P_{k_1} = P_{k_2} = \cdots = P_{k_\alpha}$, then an additional tie-breaking rule in selecting $k$ is required. We choose the index $k_i$ such that

$$k_i = \text{Argmin}_{\,q\in C} \min \left\{D_q^*, U_q^*\right\} . \tag{12}$$

If we still have more than one $k_i$ satisfying (12), then the smallest such index, which is uniquely defined, will be picked.

Penalties are also used as a device to accelerate the enumeration process. One way we calculate penalties is to perform one dual simplex iteration. This operation may or may not be able to restore primal feasibility, but it provides a lower bound for how much the objective value has to increase in order to find a feasible solution. The same consideration applies for other methods in evaluating penalties. If we can estimate this lower bound very accurately, then many nodes can be fathomed without actually investigating their descendants and hence accelerate the enumeration process.

## §4.4 Fathoming Tests

In branch-and-bound algorithms, it is very important to reduce the number of potential branches. Using integer programming terminology, we say that we want to *fathom* a branch as quickly as possible. Various fathoming tests are available and following are the ones adopted in the algorithm.

**Fathoming Test 1 :** Suppose that the incumbent and the LP-relaxation of the current subproblem have objective values $Z_{opt}$ and $Z$ respectively. This branch is fathomed if

$$Z + \min\left(D_q^*, U_q^*\right) \geq Z_{opt} \tag{13}$$

for any GCC which has not been partitioned yet. It is clear that if (13) is satisfied, no better solution can be found along this branch.

**Fathoming Test 2 :** Let $F_i$ be the set of free variables in the $i$th SOS and $V$ be the set of variables already fixed at value 1. If there exist a $k$, $1 \leq k \leq l_1$, such that

$$\sum_{j \in V} a_{kj} + \sum_{i=1}^{m} \max\left(a_{kj}, j \in F_i\right) < b_k,$$

or

$$\sum_{j \in V} c_{kj} + \sum_{i=1}^{m} \min\left(c_{kj}, j \in F_i\right) \geq Z_{opt},$$

then the branch is fathomed. Either there is no feasible completion or no better solution exists.

**Fathoming Test 3 :** If constraint (1) has not been partitioned and a 1-branch is required, then

$$F_1 \bigcap T_1 = \emptyset \tag{14}$$

implies that the branch is fathomed. Similarly, if a 0-branch is required, then

$$F_1 \bigcap T_1 = F_1 \tag{15}$$

indicates that there is no need to partition further.

A by-product of fathoming tests is the *forced* branch. A forced branch can accelerate the algorithm in finding an optimal solution because it eliminates unpromising branches. For example, if $D_q^* = \infty$ and (13) is not satisfied, then we have to make the 1–branch and need not consider the alternative branch. There are other instances where a forced branch exists. Basically, we check whether there are unpartitioned GCC's in which the set of primary (secondary) variables is a subset or superset of the set of corresponding free variables. Suppose that the constraint set contains contingent constraints

$$\sum_{j \in G_i} x_j - \sum_{j \in H_i} y_j \geq 0, \qquad \text{for} \quad i = 1, 2, 3$$

where $x_j$'s are variables from the first SOS and $y_j$'s are from the $m$th SOS. $G_i$ and $H_i$ are subsets of their respective SOS's. Consider the case where we make the 0–branch for constraints 1 and 2; then A$= Z_{n_1} - (G_1 \bigcup G_2)$ represents the set of free variables for the first SOS. If $A \subset G_3$, then the third constraint must make a 1–branch. In particular, if $|A| = 0$, then the branch is fathomed; and if $|A| = 1$, we can fix the only remaining free variable at value 1. If $A \bigcap G_3 = \emptyset$, then the 0–branch for the third constraint is required. We also have $\sum_{j \in H_1} y_j = 0$ and $\sum_{j \in H_2} y_j = 0$, so the same argument goes through if we define A$= Z_{n_m} - (H_1 \bigcup H_2)$. In addition, if $A \subset H_3$, then $\sum_{j \in G_3} x_j = 1$. More elaborate uses of the above procedure can be found in the examples of Section 4.6.

To summarize, steps G4, G5, and G6 in the algorithm G can be written as

**Procedure** B(*Refinement of the Branch Procedure* ).This procedure is the detailed statement of steps G4–G6 in the algorithm G.

**B1.** [GCC exhausted?] If all GCC are exhausted, go to G8.

**B2.** [Forced Branch?] If there is no forced branch, go to B4.

**B3.** [Fathom?] Fathoming tests. If fathomed, go to G3. Otherwise, update list map M and go to B1.

**B4.** [Penalty.] Compute penalties.

**B5.** [Fathom?] Perform fathoming tests. If fathomed, go to G3.

**B6.** [Branch.] Select constraint to partition next. Update list map M. Go to G7.

## §4.5 Handling Binary Data

Another advantage of the proposed algorithm is that the computer core required to store the group contingent constraints is less than for the general constraints, which upgrades the computational efficiency. For any specific group contingent constraint, its coefficients can take value 0 or 1 in the SOS which contains primary variables, take value 0 or —1 in the SOS which contains secondary variables, and take value 0 otherwise. Hence we can use a *bit* instead of a *computer word* to record these coefficients. Table 4–1 illustrates how this is done.

Table 4–1 : Bit Map for Group Contingent Constraints

| GCC No. | Primary Var. in SOS No. | Secondary Var. in SOS No. | Bit Map for Primary Var. | Bit Map for Secondary Var. |
|---|---|---|---|---|
| 1 | 3 | 1 | 10100 | 00101 |
| 2 | 2 | 3 | 10111 | 01011 |
| ⋮ | ⋮ | ⋮ | ⋮ | ⋮ |

The first column in Table 4–1 indicates which GCC is under consideration. The second and the third column record the special ordered sets involved. We number the bits in a word ‡ as in Table 4–2.

Table 4–2 : Bits Representation of a Word

|   |   |   |   |   |   |   |   |   |
|---|---|---|---|---|---|---|---|---|
| — | — | — | 6 | 5 | 4 | 3 | 2 | 1 |

‡ In the computer science literature, bits are numbered in the reverse order and the first bit is bit 0.

To simplify our discussion, we assume that the number of bits in the computer word under consideration is greater than the number of variables in the SOS. The last two columns of Table 4-1 represent the coefficients of group contingent constraints . The $r$th bit of the word takes the value 1 if the $r$th variable in the corresponding SOS is a primary or secondary variable. Specifically, the primary variables, represented in the first row of Table 4-1, are in the third SOS, and the secondary variables are in the first SOS. The GCC it represents is

$$x_{33} + x_{35} - x_{11} - x_{13} \geq 0.$$

Similarly, the second row summarizes the constraint

$$x_{21} + x_{22} + x_{23} + x_{25} - x_{31} - x_{32} - x_{34} \geq 0.$$

It is clear that the required storage space can indeed be reduced.

We can also accelerate the algorithm by using logic operations instead of arithmetic operations to conduct certain fathoming tests. Let W be a mapping from the set of indices to a computer word such that the $r$th bit of the word is of value one if $r$ is in the index set, and zero otherwise. Now, (14) can be expressed as

$$W(F_1) \odot W(T_1) = 0,$$

and (15) becomes

$$W(F_1) \odot W(T_1) = W(F_1)$$

where $\odot$ stands for the boolean AND operator which operates on computer words bit by bit. To test whether $j \in F_i$, one can ask if

$$W(\{j\}) \odot W(F_i) = 0.$$

Other index manipulations can also be performed by boolean algebra and hence improve the efficiency of the algorithm.

## §4.6 Examples

We present two examples to demonstrate how the proposed algorithm works. The first example illustrates the flow of the algorithm. The second example shows how logical relationships among contingent constraints can be utilized to fathom unpromising branches.

**Example 1 :**

This is a knapsack MCIP problem with binary-valued constraints. A single general type constraint often occurs, for example, when we have budgetary restrictions. Suppose that we want to

$$\begin{array}{lrl}
\text{Minimize} & x_1+3x_2+4x_3+2y_1+3y_2+4y_3+2z_1+3z_2+4z_3 & \\
\text{subject to} & 2x_1+5x_2+9x_3+y_1+2y_2+3y_3+3z_1+4z_2+5z_3 \geq 12 & \\
 & x_3 - y_3 \geq 0 & \text{(i)} \\
 & y_1 - z_3 \geq 0 & \text{(ii)} \\
 & x_1+x_2+x_3 = 1 & \\
 & y_1+y_2+y_3 = 1 & \\
 & z_1+z_2+z_3 = 1 &
\end{array}$$

where all variables are binary variables. To simplify our presentation, we replaced the branching scheme (G5) by a straightforward selection criterion. We will bring back the BVC's according to the order they appeared in the original system. In addition, we always undertake the 0-branch first. A step by step account of the procedure follows.

[G0] Solve the LP-relaxation of (R). Non-zero values are $x_1 = \frac{1}{7}$, $x_3 = \frac{6}{7}$, $y_1 = 1$, $z_1 = 1$, and $Z = \frac{53}{7}$.

[G1] Certain variables take fractional values.

[G4] BVC's have not been exhausted.

[G5] Execute 0–branch on (i), i.e., $x_3 = 0$.

[G6] Current branch can not be fathomed yet.

[G7] Reoptimize. Non–zero values are $x_2 = 1$, $y_2 = 1$, $z_3 = 1$, and Z=10.

[G1] Current solution contains integers only.

[G2] Current solution is not feasible.

[G5] Perform 0–branch on (ii), i.e., $y_1 = 0$.

[G6] Fathomed. By setting $x_3 = y_1 = y_3 = z_3 = 0$, the general constraint can never be satisfied.

[G3] Update map M.

[G9] Map M is not empty.

[G5] Execute 1–branch on (ii), i.e., $y_1 = 1$.

[G6] Fathomed. $x_3 = 1$ and $y_1 = 1$ imply that the general constraint can never be satisfied.

Repeat steps [G3], [G9].

[G5] Perform 1–branch on (i), i.e., $x_3 = 1$.

[G6] Current branch can not be fathomed.

[G7] Reoptimize. Non–zero values are $x_3 = 1$, $y_1 = 1$, and $z_1 = 1$.

[G1] All variables take integer values.

[G2] Current solution is feasible.

[G5] Update incumbent $Z_{opt} = 8$. Update map M.

[G9] Terminated.

In this particular example, the time–consuming step G8 was never executed.

**Example 2 :**

In this example, we want to demonstrate how the logical operations are performed. The problem is

$$\begin{array}{lll}
\text{Minimize} & x_1+2x_2+4x_3+2y_1+5y_2+6y_3+9y_4+9z_1+10z_2+10z_3 & \\
\text{subject to} & x_1 + x_3 - y_2 - y_3 - z_2 \geq 0 & \text{(i)} \\
& -x_2 + y_3 - z_1 - z_3 \geq 0 & \text{(ii)} \\
& -x_3 - y_1 - y_4 + z_3 \geq 0 & \text{(iii)} \\
& x_1 + x_2 + x_3 = 1 & \\
& y_1 + y_2 + y_3 + y_4 = 1 & \\
& z_1 + z_2 + z_3 = 1 &
\end{array}$$

where all variables are binary variables. There is no general constraint in this example and the LP-relaxation of (R) can be solved by inspection. We again adopt the selection criterion described for Example 1 at step G5.

[G0] Solve the LP-relaxation of (R). Non-zero values are $x_1 = 1$, $y_1 = 1$, and $z_1 = 1$.

[G1] All variables take integer values.

[G2] Solution is not feasible.

[G5] Execute 0-branch on (i), i.e., $x_1 + x_3 = 0$.

[G6] $x_2 = 1$ (since $x_1 + x_2 + x_3 = 1$), and $y_2 = y_3 = z_2 = 0$ (from (i)).

Since $x_2 = 1$, we can only partition (ii) on 1-branch.

By Fathoming test 3, this is not possible. This branch is fathomed.

[G3] Update map M.

[G9] Map M is not empty.

[G5] Execute 1-branch on (i), i.e., $x_1 + x_3 = 1$.

[G6] Unable to fathom.

[G7] Reoptimize. Solution remains unchanged.

Repeat steps [G1], [G2].

[G5] Perform 0–branch on (ii), i.e., $y_3 = 0$.

[G6] $z_1 + z_3 = 0$ (from (ii)) implies $z_2 = 1$ (since $z_1 + z_2 + z_3 = 1$).

$z_2 = 1$ implies $y_2 + y_3 = 0$ (from (i)).

$y_2 + y_3 = 0$ implies $y_1 + y_4 = 1$ ($\sum_{i=1}^{4} y_i = 1$).

$y_1 + y_4 = 1$ forces $z_3 = 1$ (from (iii)).

This branch is fathomed by Fathoming test 3.

Repeat steps [G3], [G9].

[G5] Execute 1–branch on (ii), i.e., $y_3 = 1$.

[G6] Unable to fathom.

[G7] Reoptimize. Non–zero values are $x_1 = 1$, $y_3 = 1$, and $z_1 = 1$.

[G1] All variables take integer values.

[G2] Solution is feasible.

[G3] Update incumbent $Z_{opt} = 16$. Branch fathomed automatically. Update map M.

[G9] Terminated.

Logical tests are very important for the fathoming tests involving BVC's. An exhaustive test of logical relationships among all BVC's can be very expensive. Compromises must be made when deciding which of the possible logical tests should be conducted.

# CHAPTER 5

# HEURISTIC ALGORITHMS

*Exact algorithms for integer programming problems often require substantial computational effort in order to locate and verify an optimal solution. Heuristic (approximate) procedures are designed to produce satisfactory solutions of the IP problems with much less computational work. Three different heuristic approaches are presented in this Chapter.*

## §5.1 Introduction

The efficiency of a branch and bound (BB) algorithm usually deteriorates very fast as the number of integer variables increases. Many problems simply are too large to be solved exactly. Even special-purpose BB algorithms are not able to solve some large sized problems within reasonable computer time. The class of NP-complete problems serves as a good example. Many well-known problems belong to this class, e.g., the traveling salesman problem, and the set partitioning problem [Garey and Johnson (1978), and Savage (1976)]. One can expect the solution times for these problems to increase exponentially as a function of the number of integer variables. The MCIP problems are also NP-complete [Martin, 1980]; hence, it is important to have good heuristic procedures that can locate good solutions of large sized problems very quickly.

Besides the computational difficulty mentioned above, there are other reasons that prompted the development of heuristic algorithms. A decision-maker sometimes prefers to have several *good* solutions rather than a guaranteed optimal solution. Recall that a mathematical model approximates a real-world problem and an optimal solution provides only limited information. It is possible that the optimal solution violates certain considerations that can not be captured explicitly by the model. In addition, the optimal solution may be very sensitive with respect to certain unknown parameters and the decision-maker would be reluctant to adopt such an unstable solution. If there are several good solutions available, further analyses can be conducted to evaluate the risks and the uncertainties associated with these solutions and the decision-maker can then select the best solution available. Many heuristic procedures produce a number of good solutions within very reasonable computer time and so may serve this end very well. Another reason is that some problems require solutions on a real-time basis. The exact algorithm, even if it does not take a very substantial amount of time to find an optimal solution, may still not be the proper tool to adopt.

Heuristic procedures also are used to supplement exact algorithms. Good solutions located by heuristic algorithms serve as incumbent solutions in the exact BB algorithm, thereby accelerating the enumeration process. A simple two-phase hybrid BB algorithm emerges. The first phase is to apply a heuristic algorithm to locate good solutions, and the second phase is to use an exact approach to improve upon the solutions obtained earlier and eventually to find and verify an optimal solution. Such procedures have been adopted by Hillier (1969b).

Good solutions produced by heuristic algorithms can also be useful in cutting plane algorithms. Suppose that we have an IP problem with the objective function $c \cdot x$ which is to be minimized. If a feasible solution $x_c$ is located with the objective value $Z_{inc}$ (*inc* stands for incumbent), then the inequality

$$\mathbf{c} \cdot \mathbf{x} \leq Z_{inc}$$

can be used as a cut, or as a source row for a cut.

Heuristic procedures may be devised according to specific goals that we want to achieve. For some applications, we are interested in finding a feasible point which is reasonably good very quickly, e.g., solutions which are required on a real-time basis. For other applications, we may want the feasible solution to satisfy some requirement, e.g., the objective value associated with the solution is within a certain range of the optimal objective value. Some algorithms are designed to accommodate all the constraints very rigorously, whereas the other algorithms may allow flexibility in certain constraints. In addition, we can also take advantage of the problem structure in designing special-purpose heuristic algorithms.

Early work on heuristic procedures was done by Healy (1964), Senju and Toyoda (1968), and Trauth and Woolsey (1968). Subsequent development was carried out by Hillier (1969a,b), Toyoda (1975), and Balas and Martin (1978), among others. Additional discussion can be found in, for example, Wolsey(1980).

## §5.2 General-Purpose Heuristic Procedures

We will introduce two simple general-purpose heuristic procedures. The first procedure (H1) is actually an option which can be implemented with any exact BB algorithm. Recall that for a BB algorithm, we first locate a feasible solution and then make improvements upon this solution. For the procedure (H1), we simply restrict the number of times that an improvement is to be made. For example, we can terminate the updating process after two improvements beyond the initially found solution are observed. Procedure (H1) works best if a good solution can be located after only a few improvements and the optimality conditions are difficult to verify.

As we shall see in Chapter 6, procedure (H1) is very promising for the problems we have tested. For most of our testing problems, an optimal solution is found with just one or two improvements after we located a feasible solution. The bulk of the computational effort is to verify the optimality of the incumbent. A major drawback of this approach is that we don't know how good is the solution we found. If we have an optimal solution of the LP–relaxation of the original problem, we can compare the objective values associated with the two solutions. The difference between the two objective values is an upper bound on the error. However, this can be a very loose bound because the optimal objective values of an IP problem and its LP–relaxation can differ substantially. By setting the number of improvements allowed, we can provide a reasonable tradeoff between the computational time required and the quality of the solution. For example, if the updating process is allowed to continue sufficiently many times, an optimal solution often results.

The second heuristic procedure (H2) is so designed that it provides a statement as to the quality of the solutions found. It usually produces several fairly good feasible points. To simplify our presentation, we assume that the objective function has nonnegative coefficients and is to be minimized. Recall that for any enumeration type algorithm, one way to fathom a branch is to show that no better solutions exist along the branch. Heuristic procedure (H2) relaxes this fathoming requirement. Assume that an incumbent $\mathbf{x}_c$, with the objective value $Z_{inc}$, is available. A branch is considered to be fathomed by (H2) if the best objective value that could be attainable along this branch, $Z_l$, is not more than $p$ percent better than the value

$Z_{inc}$. This fathoming criterion may be expressed as

$$Z_l(1 + p\%) \geq Z_{inc} \tag{1}$$

and $p$ is given by the user to control the accuracy of the optimal solution. Note that if $Z_l$ and/or $Z_{inc}$ can take negative values, the inequality (1) will not properly represent the fathoming criterion adopted by (H2). When $p = 0$, the heuristic algorithm becomes an exact algorithm. There are many ways to evaluate $Z_l$. In the context of the proposed algorithm, the easiest way is to define

$$Z_l = Z + \min\left\{D_q^*, U_q^*\right\}$$

where $Z$, $D_q^*$, and $U_q^*$ are defined in Chapter 4. Replacing the fathoming test 1 (Chapter 4, (13)) by (1) in the MCIP/BVC algorithm, the resulting procedure is the heuristic procedure (H2).

It is easy to show that the best objective value produced by (H2) is within $p$ percent of the optimal value, as stated in Proposition 5.2.1.

**Proposition 5.2.1.** *Suppose that the objective function of (P) has nonnegative coefficients and is to be minimized. Then the best solution of (P) attained by the procedure (H2) yields an objective value which is within p percent of the optimal objective value of (P).*

*Proof.* If (H2) produces no feasible solution, then the original problem is infeasible and the proposition is trivally true.

Suppose that the best solution produced by (H2), $\mathbf{x}_{inc}$, has the objective value $Z_{inc}$. Furthermore, let the objective values associated with the sequence of incumbents leading to $\mathbf{x}_{inc}$ be $Z_{inc}^1, Z_{inc}^2, \cdots, Z_{inc}^k = Z_{inc}$. This is a monotone decreasing sequence. For any optimal solution with the objective value $Z_{opt}$, either

$Z_{opt} = Z_{inc}$, in which case the proposition is true, or it is fathomed by (H2) for some $Z^t_{inc}$, $1 \leq t \leq k$, such that

$$Z_l(1 + p\%) \geq Z^t_{inc}$$

where $Z_l$ is defined earlier and $Z_{opt} \geq Z_l$. Consequently, we have

$$Z_{opt}(1 + p\%) \geq Z_l(1 + p\%) \geq Z^t_{inc} \geq Z_{inc},$$

which concludes the proof. ∎

Testing results of (H1) and (H2) are in Chapter 6. Both procedures can be easily incorporated with any exact BB algorithms. Thus, we call them general-purpose heuristic procedures.

## §5.3 Special-Purpose Heuristic Procedure

Heuristic procedures that are designed based upon certain assumptions and to meet specific goals are termed special-purpose heuristic procedures. These procedures take advantage of the special structure of the problem as well. In this section, we will develop a special-purpose heuristic procedure which *solves* the MCIP/BVC problem. We first state the basic assumptions imposed and then describe the algorithm in the context of a capital budgeting problem. The computational results are in Section 6.5.

It is often difficult to state the constraints precisely when we formulate a real-world problem, especially if the constraints represent restrictions in the uncertain future. The basic assumption of the proposed special-purpose heuristic procedure is that we allow some flexibility in the constraints. In addition, we would like the algorithm to provide a wide range of potential alternatives by having multiple *solutions*. The procedure is a Lagrange based approach. Refer to Geoffrion (1974) for a good discussion of this approach as applied to integer programming problems. A sequence of simplified MCIP/BVC problems is generated and each problem may produce a good solution. The procedure terminates when certain stopping criteria are met.

The standard formulation $(P)$ of the MCIP/BVC problem assumes that the objective function is to be minimized. In order to interpret the example given below in a more natural way, we convert $(P)$ from a minimization form to a maximization form. Coupled with other minor modifications in formulating the general constraints, the new problem $(P')$ becomes

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} \\ \text{subject to} & Ax \le b \\ & Gx \ge 0 \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1, 2, ..., m \\ & x_{ij} \in \{0, 1\} \quad \forall\ i, j \end{array} \tag{$P'$}$$

where $A$ and $G$ are matrices with appropriate dimensions and $b = (b_1, b_2, \cdots, b_{l_1})^T$ is a column vector of dimension $l_1$.

We use a capital budgeting problem to illustrate the objectives that the algorithm is designed to achieve. Suppose that we are to develop an investment strategy, i.e., to select a good investment portfolio among the potential projects, for the next $l_1$ years. Besides the multiple choice and the contingent properties that exist among the projects, we assume that the only other constraints are the budget restrictions. Referring to the formulation $(P')$, the general constraints $Ax \le b$ would represent the annual budget restrictions where $b_1$ is the budget for the current year and the remaining $b_i$'s are budgets for future years. An important characteristic of the problem is the uncertainty pertaining to the right-hand-side coefficient vector $b$. The budgets for future years are only projected values and the decision-maker may actually have some controls over these figures. A solution which does not require unreasonable annual expenditures beyond the projected budgets may be considered as an *acceptable* solution of $(P')$ and hence a viable proposal. This fact suggests the desirability of having multiple solutions since these acceptable solutions can satisfy the budget restrictions differently and each solution may enjoy certain advantages over other solutions. The final decision rests on the decision-maker's judgement. The optimal solution obtained by the MCIP/BVC algorithm, or other algorithms, plays a very restrictive role insofar as the decision-maker is concerned.

As we shall see later in Section 6.2, the MCIP/BVC algorithm performs extremely well when there is only one general constraint. Its performance deteriorates as the number of general constraints increases. Together with the fact that multiple solutions are desired, a Lagrange based heuristic approach is proposed. It incorporates the Lagrange multipliers in a specific way which accommodates the above objectives and efficiently utilizes the MCIP/BVC algorithm. The multipliers can also be used in ways other than what we describe next, and different objectives may then be accomplished.

Before applying the proposed procedure, we first select one general constraint out of the $l_1$ general constraints. All the solutions found are required to satisfy this selected constraint but none of the other general constraints. In the capital budgeting example provided earlier, this constraint may correspond to the budget restriction for the current year. (If none of the original general constraints had been required to be satisfied precisely, we may add a constraint to the system and consider it to be the constraint that every solution needs to satisfy. For example, we could add a constraint that represents the restriction of the overall spending level over the planning horizon.) The procedure begins by replacing the existing objective function by a weighted sum of the original objective row and the general constraints that are not required to be satisfied. We then eliminate these general constraints from consideration. The resulting problem possesses the knapsack MCIP/BVC structure which can be solved very efficiently. The weights associated with the general constraints are then updated and the process repeats itself. This iterative scheme will be terminated when certain stopping criteria are met.

For simplicity, we assume that the first general constraint of $Ax \leq b$ is the one that is required to be satisfied at all times. Let $a_i$ be the row vector that corresponds to the $i$th row of the matrix $A$. Denote

$$A' = \begin{pmatrix} a_2 \\ a_3 \\ \vdots \\ a_{l_1} \end{pmatrix},$$

so then the problem $(P')$ can be written as

$$\begin{array}{ll} \text{Maximize} & \sum_{i=1}^{m} \sum_{j=1}^{n_i} c_{ij} x_{ij} \\ \text{subject to} & a_1 x \leq b_1 \\ & A'x \leq b' \\ & Gx \geq 0 \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1, 2, \dots, m \\ & x_{ij} \in \{0, 1\} \quad \forall\, i, j \end{array}$$

where $b' = (b_2, b_3, \cdots, b_{l_1})^T$ is a column vector and $A'$ and $G$ are matrices of appropriate dimensions. The Lagrange based approach can be explained more readily by first forming the Lagrangian dual problem $(D)$,

$$\begin{array}{lll} \text{Minimize}_{\,U} \text{ Maximize}_{\,x_{ij}} & \sum_{i=1}^{m}\sum_{j=1}^{n_i} c_{ij}x_{ij} - U(A'x - b') & \\ \text{subject to} & a_1 x \le b_1 & \\ & Gx \ge 0 & \\ & \sum_{j=1}^{n_i} x_{ij} = 1 \qquad i = 1,2,\dots,m & (D) \\ & \mu_k \ge 0 \qquad k = 2,3,\dots,l_1 & \\ & x_{ij} \in \{0,1\} \quad \forall\, i,j & \end{array}$$

where $U = (\mu_2, \mu_3, \dots, \mu_{l_1})$ consists of the Lagrangian multipliers. Note that $x_{ij}$'s are still required to be integers and they are variables in both $(P')$ and $(D)$. Define $D(U^t)$ to be the reduced dual problem where the Lagrangian multipliers in $(D)$ are replaced by a known vector $U^t = (\mu_2^t, \mu_3^t, \cdots, \mu_{l_1}^t)$. Let $Z_D$ and $Z_{D(U^t)}$ be the optimal objective values of the problems $(D)$ and $D(U^t)$, respectively.

The underlying rationale of the procedure is to solve a sequence of the reduced dual problems $D(U^t)$. It begins by solving the problem $D(U^1)$ for a given $U^1$. It is clear that the minimization part of $D(U^1)$ is redundant and the resulting problem is indeed a knapsack MCIP/BVC problem. We then check whether the solution $x^t$, which is an optimal solution found at the $t$th iteration, is an acceptable solution with respect to $(P')$. Specifically, we consider the solution $x^t$ to be acceptable if for each general constraint $a_k x \ge b_k$, $k = 2, 3, \dots, l_1$, the inequality

$$a_k \mathbf{x}^t \ge b_k(1 - p\%)$$

holds for a given $p$. If it is acceptable, we record it for future reference and a check for termination is executed. The procedure repeats itself for a new $U$ if the stopping criterion is not met. Typically, the initial $U^1$ vector is a zero vector or assumes the values of the dual variables associated with an optimal solution of the LP–relaxation of $(P')$. Successive $U$ vectors are obtained according to the standard subgradient method:

$$\mu_i^{t+1} = \mu_i^t - S^t(a_i x^t - b_i), \qquad i = 2, 3, \ldots, l_1 \tag{2}$$

where $t$ is the iteration counter which starts with 1 and $S^t$ is a positive scalar step size at the $t$th iteration. The step size $S^t$ is defined by

$$S^t = \frac{W(Z_{D(U^t)} - \underline{Z})}{\|A'x^t - b'\|} \tag{3}$$

where $\|v\|$ represents the Euclidean norm of the vector $v$, $W$ is the relaxation coefficient which satisfies $0 < W \leq 2$, and $\underline{Z}$ is a lower bound on $Z_D$. We will not discuss the subgradient method in great length here. The interested reader should refer to Crowder, Held, and Wolfe (1974) for the theoretical and computational aspects of the approach.

It is well-known that the optimal objective value of $(P')$, $Z$, is less than or equal to $Z_D$. Their difference, $(Z_D - Z)$, is often referred to as the duality gap. The duality gap usually is a strictly positive number in integer programming. From the inequalities

$$Z \leq Z_D \leq Z_{D(U^t)}, \tag{4}$$

it is clear that the objective value of any feasible solution of $(P')$ can serve as a lower bound $\underline{Z}$ of $Z_D$. The initial $\underline{Z}$ can be defined as the sum of the objective coefficients that are the smallest in each SOS. This objective value may not correspond to a feasible solution of $(P')$, but it is the smallest objective value that $(P')$ can achieve. Subsequently, $\underline{Z}$ is updated by any feasible solution of $(P')$ which yields a larger objective value.

The expression $(Z_{D(U^t)} - \underline{Z})$ in the nominator of (3) represents the amount that the current optimal dual objective value can be decreased. The denominator of (3) can be zero only if the solution obtained at the previous iteration is an optimal solution of $(P')$. A solution which makes the denominator of (3) zero implies that it is a feasible solution of $(P')$. In addition, it produces the same objective value for the problems $D(U^t)$ and $(P')$. Using (4) and the fact that $Z$ is the largest objective value of $(P')$, we have $Z = Z_D$ and the solution obtained earlier must already be an optimal solution of $(P')$.

In the current implementation, the iterative scheme will be terminated if one of the following three stopping rules is met. The stopping rules are

(a) When the objective values $Z_{D(U^t)}$ and $\underline{Z}$ become close enough,

(b) When the number of iterations reaches a prescribed limit, and

(c) When a given number of acceptable solutions is found.

The stopping rule (a) can be expressed as

$$\underline{Z} \geq (1 - q\%) Z_D^* \tag{5}$$

where $Z_D^*$ is the best objective value of the dual problem $(D)$ (the smallest $Z_{D(U^t)}$) found so far. The iterative scheme is terminated when (5) holds for a given $q$. Stopping criteria (b) and (c) are technical provisions which guarantee that the process has a finite termination. The step-by-step description of the procedure (H3) is given below.

**Algorithm H3** (*Heuristic* ). This algorithm is designed as a special-purpose heuristic approach to solve MCIP/BVC problems.

**A0.** [Initialization.] Let $Z_D^* = \infty$, $\underline{Z} = -\infty$, $t = 1$, and $U^1 = 0$.

**A1.** [Solve.] Solve $D(U^t)$. If $Z_{D(U^t)} < Z_D^*$, set $Z_D^* = Z_{D(U^t)}$.

**A2.** [Acceptable.] Record $x^t$ if it is an acceptable solution with respect to $(P')$.

**A3.** [Feasible.] If $x^t$ is a feasible solution with respect to $(P')$ and its objective value $Z_P$ is larger than $\underline{Z}$, then set $\underline{Z} = Z_P$.

**A4.** [Terminate.] Check for the stopping criteria (a)–(c) stated above. If any one of the conditions is met, stop.

**A5.** [Update.] Set $t = t + 1$. Update $U^t$ according to (2) and (3). Go to step (A1).

When we check for termination at (A4), we use $Z_D^*$ instead of $Z_{D(U^t)}$ to see whether the criterion (a) is satisfied. The numerical assumptions on how to determine if a solution $x^t$ is acceptable and how small the duality gap should become before the process stops will be addressed in Section 6.5.

# CHAPTER 6

# COMPUTATIONAL RESULTS

*We commence this chapter by describing the testing procedure. Three sets of problems are run under various algorithms and the results are compared. We also examine the effects of different branching criteria adopted in the algorithm. Results by applying heuristic approaches are reported in the last two sections.*

## §6.1 Testing Procedure

In order to compare the relative merits of the MCIP/BVC algorithm to other algorithms, we have selected three sets of problems with which to do extensive testing. Each set of testing problems is designed to examine a particular aspect of the algorithm. The first set of problems aims to determine the overall efficiency of the algorithm and to identify the key parameters pertaining to the efficiency of the algorithm. Recall that the standard formulation takes the form

$$\begin{array}{ll} \text{Minimize} & \sum_{j=1}^{m} \sum_{k=1}^{n_j} c_{jk} x_{jk} \\ \text{subject to} & Ax \geq b \\ & Gx \geq 0 \\ & \sum_{k=1}^{n_j} x_{jk} = 1 \qquad j = 1, 2, ..., m \\ & x_{jk} \in \{0, 1\} \quad \forall\ j, k \end{array}$$

Let $a_{ijk}$ denote the coefficients of the constraint matrix $A$, where the element $a_{ijk}$ represents the entry for the $i$th constraint and the $k$th variable in the $j$th SOS. For the first set of test problems, each $a_{ijk}$ as well as $c_{jk}$ is a random number generated uniformly over a prescribed interval and the right hand side, $b_i$, is calculated by

$$b_i = \sum_j [\max_k (a_{ijk}) + \min_k (a_{ijk})]/T_i,$$

where $T_i$ is called the *tightness ratio* of the $i$th constraint. We assume that the endpoints of these prescribed intervals take nonnegative values. Furthermore, the random numbers generated are rounded to their nearest integers. These restrictions should not alter our testing results because the proposed algorithm does not take advantage of the coefficients being nonnegative. As $T_i$ increases, the right hand side decreases and the $i$th constraint becomes less restrictive since the constraint takes the form of *greater than or equal to* in the standard formulation. The binary-valued constraints are also generated randomly. For each binary-valued constraint, we first select the special ordered sets that contain primary and contingent variables, respectively. We then generate non-zero entries within each SOS. To make our comparisons easy to follow, we adopt the following values as the base case:

Number of general constraints = 3,

Number of binary valued constraints = 8,

Number of SOS = 10,

Number of variables in each SOS = 5, and

Tightness ratio = 2.0.

The total number of binary variables is 50, which is considered to be a moderate sized problem. Tightness ratios are assumed to be the same for all general constraints. These numbers will be altered in the subsequent sensitivity studies in order to show how each parameter affects the efficiency of the algorithm.

The second set of testing problems is the multiple choice knapsack problem with binary-valued constraints. For these testing problems, our goal is to determine whether the MCIP/BVC algorithm can be accelerated if the matrix $A$ possesses some special structure. In the present case, the reduced problem is a multiple choice knapsack problem, for which there are efficient algorithms available. The algorithm adopted in the current implementation can be found in Sinha and Zoltner(1979a). The testing problems are also generated randomly as described earlier but with different numerical specifications. The number of variables ranges up to 400.

The last set of problems is the National Basketball Association (NBA) scheduling problems developed by Bean (1980), as discussed in detail in the Appendix. These problems resemble the classical assignment problems to a certain extent, but the optimal solutions to their LP–relaxation do allow fractional values. The procedure used by Bean to generate these testing problems can also be found in the Appendix. Since Bean has documented computational results for solving these problems with his algorithm, our major concern is to compare the relative efficiency of his and our algorithms.

## §6.2 Testing Results

All the computational work was done at Stanford University's Sierra computer facility. Sierra is a DECSYSTEM–20. Before we present the testing results, a few words of caution seem appropriate. Sierra is a time–sharing system and requires various overhead operations, so it is very difficult to observe the precise CPU times (which include the input/output times). Depending upon the time of day, which affects the load of the system, the observed CPU times can deviate as much as 15%. All of our testing was performed in approximately the same time slot when the system load is relatively light, so we expect that the CPU time deviation was much less than 15%. Also note that each entry in the following tables is the average time of ten (10) problems, unless otherwise specified, which should lessen the effect of the randomness of the problems generated as well as the usually large variance in the computing times associated with integer programming algorithms. The source code is written in FORTRAN 66.

## §6.2.1 Problem Set 1: General Efficiency

Table 6–1 reports the overall efficiency of the MCIP/BVC algorithm along with two other approaches. Tables 6–2 through 6–6 summarize the sensitivity results by altering various parameters one at a time. CPU times are measured in seconds.

Table 6–1 : CPU Times for Different Algorithms

| # of Binary–Valued Const. | CPU Time (Branch–Bound) | CPU Time (MCIP/BVC) | CPU Time (Beale–Tomlin) |
|---|---|---|---|
| 4 | 3.36 | 1.90 | 2.68 |
| 6 | 4.53 | 2.34 | 2.89 |
| 8 | 4.21 | 3.02 | 4.76 |
| 10 | 3.65 | 3.28 | 5.02 |
| 12 | 4.95 | 3.69 | 6.06 |
| 14 | 6.87 | 4.07 | 8.24 |
| 16 | 7.34 | 4.30 | 11.16 |

Table 6–1 compares the average CPU times for the usual branch and bound algorithm, the MCIP/BVC algorithm, and the algorithm developed by Beale and Tomlin (1969) when applied to 7 different sets of 10 MCIP/BVC problems. The usual branch and bound algorithm refers to the BB algorithm, as described in Garfinkel and Nemhauser (1972), which does not take advantage of any special structure. The MCIP/BVC algorithm performs somewhat better than the other two algorithms. The percentage increases of the computing times, as a function of the number of binary–valued constraints in the system, are about the same for the BB and the MCIP/BVC algorithms, whereas it is somewhat higher for the Beale–

Tomlin approach. This latter result is counter intuitive. A possible explanation is that the computer code for the BB algorithm adopted here is an established one developed by Reardon (1974), whereas the code for the Beale-Tomlin algorithm has been written by the author with a straightforward implementation, which might downgrade its efficiency. The same argument applies to the MCIP/BVC code since it also has been written by the author with a straightforward implementation. It is possible that the performance of the MCIP/BVC algorithm can be improved if the code is carefully implemented.

Table 6-2 : CPU Times by Varying the Number of Binary-Valued Constraints

| # of Binary-Valued Const. | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) | CPU Time (Beale-Tomlin) |
|---|---|---|---|---|---|
| 4 | 0.3 | 1.6 | 5.0 | 1.90 | 2.68 |
| 6 | 0.7 | 2.0 | 7.5 | 2.34 | 2.89 |
| 8 | 1.1 | 3.1 | 11.6 | 3.02 | 4.76 |
| 10 | 1.3 | 2.7 | 15.4 | 3.28 | 5.02 |
| 12 | 0.9 | 3.6 | 23.9 | 3.69 | 6.06 |
| 14 | 1.1 | 3.0 | 26.9 | 4.07 | 8.24 |
| 16 | 0.9 | 2.3 | 29.2 | 4.30 | 11.16 |

In Table 6-2, additional information is provided to evaluate the performance of the MCIP/BVC algorithm. There are several quantities we are particularly interested in to describe the performance of the algorithm other than the overall computational time. The *number of improvements* represents the number of times beyond the initially found incumbent that a better solution than the current incumbent is located. This quantity provides us with a clue as to how fast an optimal solution can be located and a basis to judge the heuristic procedure (H1). From our testing problems, we found that on average this number is rather small (approximately one). In addition, this number stays relatively flat as the number of binary-valued constraints increases. This suggests that the procedure (H1) could be effective for this set of testing problems.

The *number of IP solved* indicates the number of times that the step G8 was executed. This quantity is crucial since the step G8 is usually the most time-consuming operation in the entire procedure. If this quantity is large, then the value of the proposed algorithm will certainly be in doubt. From the table we see that the number of IP solved is small (usually around 3) and does not necessarily increase as the number of binary-valued constraints increases. A plausible explanation is that as we have more binary-valued constraints, the general constraints become relatively less restrictive and many branches will be fathomed, either by locating a better incumbent or otherwise, at an early stage without going through the step G8.

*Branches fathomed* indicates how many branches are being explicitly examined. A small number shows that many branches are fathomed very quickly, which may in turn demonstrate that the fathoming devices are powerful. In the case that the number of binary-valued constraints is 14, the potential number of branches fathomed could be as high as $\sum_{k=1}^{14} 2^k = 32766$. The average number of branches fathomed in this case is 26.9, which is only about 0.08 % of the potential branches. Furthermore, this ratio decreases as the number of binary-valued constraints increases. This is encouraging because it indicates that the efficiency of the MCIP/BVC algorithm does not deteriorate as the number of binary-valued constraints increases. Also note that the number of IP solved is far fewer than the branches fathomed, which shows that other fathoming devices are useful.

The last two columns of Table 6-2 report the CPU times of the MCIP/BVC and

Beale–Tomlin algorithms, respectively. We will use these two algorithms in future comparisons because we feel that they are at about the same level of efficiency as far as their implementation is concerned and they both utilize the GUB algorithm.

Table 6–3 : CPU Times by Varying the Number of General Constraints

| # of General Constraints | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) | CPU Time (Beale–Tomlin) |
|---|---|---|---|---|---|
| 1 | 0.9 | 2.5 | 11.6 | 1.52 | 2.33 |
| 2 | 1.2 | 3.0 | 14.4 | 2.18 | 3.16 |
| 3 | 1.1 | 3.1 | 11.6 | 3.02 | 4.76 |
| 4 | 1.8 | 6.6 | 19.3 | 9.10 | 16.68 |
| 5 | 1.3 | 5.7 | 17.0 | 10.64 | 35.50 |
| 6 | 1.4 | 7.4 | 17.8 | 16.02 | $\geq 60$ |

Table 6–3 displays another set of sensitivity results. We systematically change the number of general constraints. It is observed that the number of improvements is still a small number, but the number of IP solved as well as the CPU times increases very rapidly as the number of general constraints increases. This leads to a conjecture that the one key factor which determines the efficiency of the MCIP/BVC algorithm could be the ratio between the numbers of binary–valued constraints and general constraints, i.e.,

$$r = \frac{\text{the number of binary–valued constraints}}{\text{the number of general constraints}} .$$

As $r$ increases, the algorithm is likely to become fairly efficient. This perhaps can be explained by the reason we provided earlier, i.e., as the number of binary–valued constraints increases, the general constraints become relatively less restrictive and many branches can often be fathomed without undertaking the expensive step G8. Also we note that when $r$ is small, one can argue that the problem becomes less structured and the special-purpose algorithm MCIP/BVC may not be the proper tool to adopt in solving the problem.

The increasing number of IP solved reflects the increases of the computing time. The number of improvements stays small, so the suggested heuristic procedure (H1) can still be effective. The Beale-Tomlin algorithm requires more CPU times on the averages in all cases and its rate of CPU time growth is also larger than that of the MCIP/BVC algorithm for these problems.

Table 6-4 : CPU Times by Varying the Number of Variables in Each SOS

| # of Var. in Each SOS | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) | CPU Time (Beale-Tomlin) |
|---|---|---|---|---|---|
| 5 | 0.6 | 2.3 | 12.0 | 3.31 | 7.71 |
| 6 | 1.4 | 4.9 | 15.9 | 6.32 | 8.12 |
| 7 | 1.3 | 4.2 | 16.1 | 7.78 | 13.16 |
| 8 | 1.1 | 4.5 | 14.4 | 8.69 | 14.28 |
| 9 | 0.9 | 5.1 | 17.7 | 8.10 | 16.04 |
| 10 | 1.1 | 3.7 | 14.1 | 5.92 | 16.83 |

Usually in integer programming problems, the number of variables plays a very important role in determining whether the problem can be solved within a reasonable computer time. The CPU time required to solve a problem tends to increase rapidly when the number of variables is increased. In Table 6-4, however, we found that this is not necessarily the case for the MCIP/BVC algorithm. As we increase the number of variables in each special ordered set, the average CPU times did not grow as we expected. There is no obvious reason which accounts for the decreases of the average CPU times in the last two rows except the randomness of the testing problems. What we can expect is that the CPU times should not increase very rapidly. Observe that by adding one binary variable into each SOS, the number of potential solutions increases

$$p = (\frac{n+1}{n})^m = (1 + \frac{1}{n})^m$$

times where $m$ is the number of SOS and $n$ is the number of variables in each SOS. As $n$ increases, $p$ becomes smaller and converges to 1. In a general BIP problem, adding $m$ binary variables will increase the number of potential solutions by $2^m$ times, which is far larger than $p$.

Again, the number of improvements remains small and the CPU times for the Beale–Tomlin approach increase somewhat faster than that for the MCIP/BVC algorithm. It seems safe to conclude that the number of variables in each SOS is not a crucial factor as to the performance of the MCIP/BVC algorithm.

In the base case, we assumed that the tightness ratios for all general constraints are the same, with the value 2. This quantity affects the right hand sides and determines the size of the feasible region. In the next sensitivity study, we shall see the effect of altering the tightness ratio.

**Table 6–5** : CPU Times by Varying the Tightness Ratio

| Tightness Ratio | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) | CPU Time (Beale–Tomlin) |
|---|---|---|---|---|---|
| 1.7 | 1.0 | 4.8 | 16.3 | 6.32/4.75 * | 18.93/12.37 * |
| 1.8 | 0.7 | 4.7 | 16.6 | 6.23 | 12.29 |
| 1.9 | 1.3 | 4.0 | 14.7 | 5.86 | 13.93 |
| 2.0 | 1.1 | 3.1 | 11.6 | 3.02 | 4.76 |
| 2.1 | 1.0 | 2.5 | 11.0 | 2.45 | 4.77 |

*The first number is the average time of the 6 feasible problems obtained by generating 10 problems. The second number is the average time of these 6 and the remaining 4 problems which are not feasible when the tightness ratio becomes too small.

Table 6–5 shows that as the tightness ratio decreases, the problem becomes more difficult to solve. On average, the number of IP that need to be solved increases as the tightness ratio decreases. This indicates that we are more likely to exhaust all the binary–valued constraints without fathoming the branch. In other words, feasible regions are determined largely by the general constraints and the binary–valued constraints become less relevant. Hence we conclude that the MCIP/BVC algorithm should become more effective as the binary–valued constraints contribute more to the determination of the feasible region. The number of improvements is still very small, which again suggests that the heuristic procedure (H1) can be an effective one.

Table 6–6 : CPU Times by Varying the Number of Special Ordered Sets While Keeping the Total Number of Variables * Fixed

| # of Special Ordered Sets | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) |
|---|---|---|---|---|
| 3 | 0.7 | 4.7 | 18.6 | 3.29 |
| 4 | 1.2 | 3.6 | 16.7 | 2.92 |
| 5 | 1.6 | 4.8 | 17.4 | 4.04 |
| 6 | 1.1 | 3.0 | 13.4 | 3.97 |
| 10 | 1.4 | 4.9 | 15.9 | 6.64 |
| 12 | 0.6 | 4.3 | 15.9 | 6.40 |
| 15 | 0.5 | 3.1 | 12.9 | 7.10 |
| 20 | 0.8 | 3.0 | 11.7 | 7.98 |

*Total number of variables = 60.

Finally, we investigate how the number of special ordered sets affects the performance of the MCIP/BVC algorithm. To achieve this goal, we alter the number of SOS while keeping the total number of binary variables fixed. This will also change the number of variables in each SOS. Note that the number of variables in each SOS varies inversely with the number of SOS.

Table 6–6 demonstrates the case with 60 binary variables. As the number of SOS increases, the CPU time increases. It seems that the number of SOS plays a more important role than the number of variables in each SOS. In fact, we can argue that the problem becomes less structured when we have too many SOS, so that the efficiency of the proposed algorithm deteriorates. However, the computational results are not very conclusive. The difficulties of the problem raised by introducing more SOS can often be offset by the reduced number of variables in each SOS. The randomness of the testing problems here may play a more crucial factor than that in other testing results.

From the extensive testing we have conducted (over 250 testing problems in different sizes), we conclude that the MCIP/BVC algorithm generally is reasonably efficient on MCIP/BVC problems of moderate size, especially when the ratio $r$ is fairly large. In the next two subsections, we will test its performance on problems which possess other special structure.

### §6.2.2 Multiple Choice Knapsack Problem with BVC

The second set of testing problems is the multiple choice knapsack problem with binary–valued constraints. If we set aside the binary–valued constraints, the reduced problem is a binary multiple choice knapsack problem, for which there are very efficient algorithms available. Recall that the most time–consuming operation in the MCIP/BVC algorithm is usually step G8. Having the special structure at hand, we can substantially reduce the CPU times required at step G8 and so improve considerably the overall efficiency. Again we assume that the number of binary–valued constraints is 8 and the tightness ratio is 2.

Table 6-7: CPU Times for Knapsack Problems with Binary-Valued Constraints

| | Number of Variables in Each SOS | | | | | |
|---|---|---|---|---|---|---|
| No. of SOS | 10 | 12 | 14 | 16 | 18 | 20 |
| 10 | 2.05 | 2.84 | 2.35 | 3.24 | 2.83 | 3.56 |
| 12 | 2.56 | 2.81 | 3.69 | 4.24 | 4.34 | 5.49 |
| 14 | 3.29 | 3.33 | 3.89 | 5.36 | 4.70 | 5.38 |
| 16 | 4.75 | 4.20 | 4.41 | 5.32 | 6.54 | 8.28 |
| 18 | 4.30 | 5.18 | 6.22 | 8.45 | 7.12 | 7.85 |
| 20 | 4.87 | 5.35 | 6.95 | 9.28 | 9.69 | 9.27 |

Table 6-7 investigates the CPU times associated with the different numbers of special ordered sets and the different numbers of variables in each SOS. The total number of variables can be as high as 400 and, on average, less than ten seconds of CPU time are required. Furthermore, the rate of growth of CPU time seems to be approximately linear with the number of variables in each SOS and only slightly more than linear with the number of SOS. This encouraging performance suggests strong potential for the algorithm in cases where special structure that can be exploited is present in the reduced problem.

The merits of the MCIP/BVC algorithm are demonstrated very favorably by this set of testing problems because step G8 consumes a relatively small portion of the total computational time. Table 6-8 provides detailed information regarding the performance of the MCIP/BVC algorithm on problems of this type when the number of binary-valued constraints varies. In each case, the number of SOS is 10 and there are 10 variables in each SOS, so the total number of (binary) variables is 100. We first note that the number of improvements remains small in all cases, which suggests that the heuristic procedure (H1) will be effective. The number of branches fathomed increases very slowly in absolute terms, and decreases in percentage terms, when the number of binary-valued constraints increases. In the case of 16 binary-valued constraints, only 0.03 percent ($40.2/\sum_{k=1}^{16} 2^k$) of the potential branches are investigated explicitly before we find and verify an optimal solution. Figure 6-1 plots the total CPU times versus the number of binary-valued constraints and we see that the CPU times increase approximately quadratically.

**Table 6-8** : CPU Times by Varying the Number of Binary-Valued Constraints in Knapsack Problems

| No. of Binary Valued Const. | Number of Improvements | # of IP Solved | Branches Fathomed | CPU Time (Seconds) |
|---|---|---|---|---|
| 4 | 0.6 | 1.5 | 5.2 | 5.14 |
| 6 | 1.0 | 2.0 | 7.8 | 6.07 |
| 8 | 0.8 | 2.7 | 11.0 | 6.86 |
| 10 | 1.3 | 3.2 | 15.9 | 8.76 |
| 12 | 1.7 | 3.5 | 21.9 | 10.57 |
| 14 | 2.0 | 4.7 | 33.4 | 13.34 |
| 16 | 2.2 | 4.9 | 40.2 | 15.14 |

### §6.2.3 NBA Scheduling Problem

The last set of testing problems involves the NBA scheduling problem. A detailed description of the problem can be found in the Appendix. There are no general constraints in these problems. The group contingent constraints are of type SM, which complicates the enumeration process. The IP problems that need to be solved at step G8 are trivial. We only need to assign the value 1 to each variable that is still a free variable in the SOS and has the smallest cost coefficient among the remaining free variables in the SOS.

Table 6-9 summarizes the relevant results. Each entry is again the average of

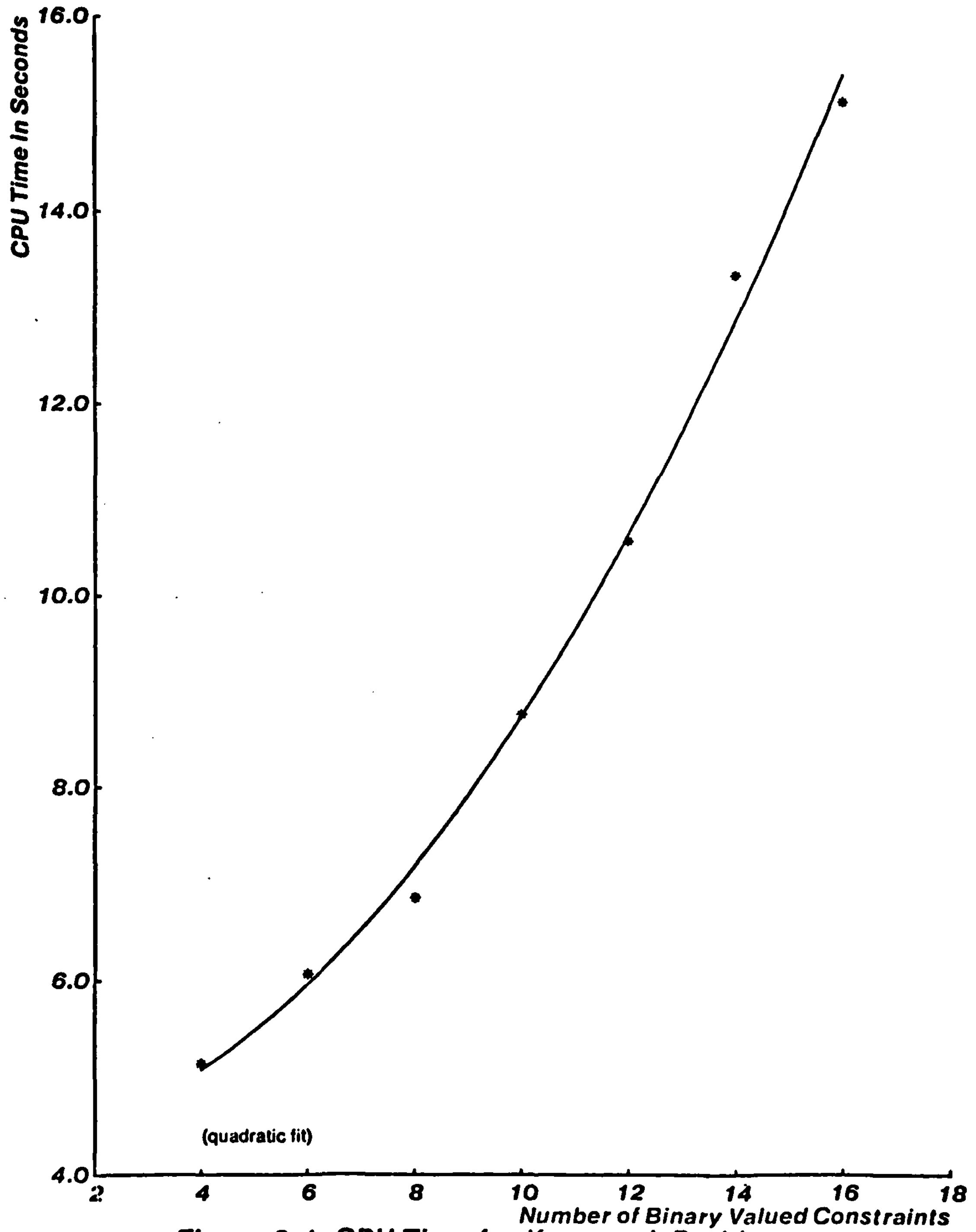


Figure 6-1: CPU Time for Knapsack Problems

10 problems, if applicable. First we note that the number of improvements is very small, which indicates that most of the computational effort is spent on verifying optimality if the initial trial solution and improvements can be quickly located. It also indicates that the heuristic procedure (H1) is worth exploring. The range of the CPU times is large, which usually happens in IP algorithms and also accounts for the large standard deviation obtained. The CPU times reported by Bean(1980) appear in the last column. We find that the average CPU times in the MCIP/BVC algorithm are much better than those obtained by Bean's algorithm on the same computer. Figure 6-2 shows that the CPU times increase approximately quadratically as the number of teams being scheduled increases.

**Table 6–9** : CPU Times for Scheduling Problems

| # of Teams Selected | Number of Improvements | Range of CPU Time (Seconds) | Average CPU Time | Standard Deviation | CPU Time (Bean) |
|---|---|---|---|---|---|
| 10 | 0.1 | 2.7– 4.4 | 3.06 | 0.47 | 12.1 |
| 12 | 0.1 | 4.1– 9.0 | 5.22 | 1.38 | 19.4 |
| 14 | 0.7 | 7.4–15.8 | 9.55 | 2.68 | 36.9 |
| 16 | 0.3 | 12.3–29.3 | 15.57 | 5.13 | 65.4 |
| 18 | 0.4 | 17.7–47.2 | 23.43 | 9.59 | 88.5 |

## §6.3 Effect of Different Branching Criteria

In Chapter 4, we introduced the branching heuristic based on choosing the worst alternative. It is a conservative measure because the rationale is to postpone the search on potentially bad branches rather than to locate good solutions which are likely to be nearby optimal. Two alternative selection criteria are presented in this section and the computational results on all three criteria are reported in Table 6–10.

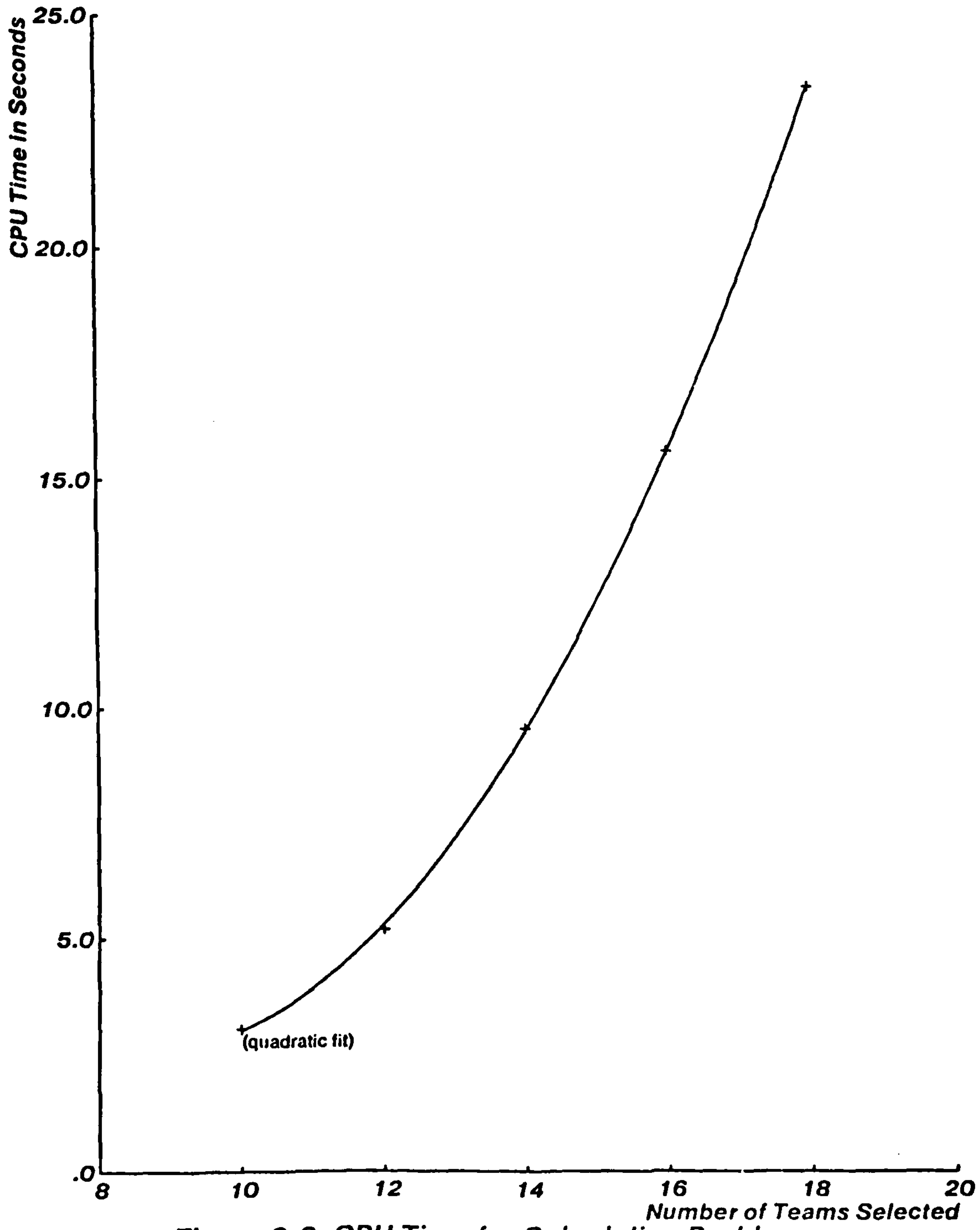


Figure 6-2: CPU Time for Scheduling Problems

Using the notation described in Chapter 4, the alternative criterion 1 is defined as

(A) Introduce the $k$th BVC where

$$k = \text{Argmax}_q \ \max \{D_k^*, U_k^*\}, \quad P_k = \max \{D_q^*, U_q^*\}.$$

(B) If the penalty $P_k = D_k^*$, execute a 0–branch. Otherwise, execute a 1–branch.

If the index $k$ is not unique, the smallest such $k$ is picked. By selecting the branch with the largest penalty, we hope to fathom the branch very quickly.

The alternative criterion 2 is defined as

(A) Introduce the $k$th BVC where

$$k = \text{Argmin}_q \ \min \{D_k^*, U_k^*\}, \quad P_k = \min \{D_q^*, U_q^*\}.$$

(B) If the penalty $P_k = D_k^*$, execute a 0–branch. Otherwise, execute a 1–branch.

Again, if the index $k$ is not unique, we select the smallest such $k$. This is an aggressive selection criterion. The rationale is to pick the branch that is most likely to contain good solutions. By finding a good solution very quickly, we hope to expedite the fathoming process for the remaining branches.

Table 6–10: Comparison of Different Selecting Criteria

| No of | Old Criterion | | Alt. Criterion 1 | | Alt. Criterion 2 | |
|---|---|---|---|---|---|---|
| BVC's | Brh. Fathom | CPU | Brh. Fathom | CPU | Brh. Fathom | CPU |
| 4 | 5.0 | 1.90 | 5.4 | 2.06 | 5.0 | 1.94 |
| 6 | 7.5 | 2.34 | 10.4 | 2.71 | 8.5 | 2.62 |
| 8 | 11.6 | 3.02 | 16.2 | 3.30 | 12.2 | 3.09 |
| 10 | 15.4 | 3.28 | 27.3 | 4.63 | 16.0 | 3.30 |
| 12 | 23.9 | 3.69 | 46.0 | 6.31 | 24.6 | 3.78 |
| 14 | 26.9 | 4.07 | 52.4 | 7.04 | 28.1 | 4.27 |
| 16 | 29.2 | 4.30 | 75.7 | 9.01 | 33.0 | 4.89 |

The problems tested are the base case problems with the number of binary-valued constraints varied (as originally used for Table 6-2). From Table 6-10, we see that the original criterion and the alternative criterion 2 produced very similar results, with slightly higher figures for the latter criterion. Because the CPU times cannot be observed precisely and the problems are random in nature, we consider these differences to be statistically insignificant. The alternative criterion 1 generated substantially higher figures, especially when the number of BVC becomes large. We conclude that the alternative criterion 1 is inferior to the other two criteria. No further tests will be conducted regarding this subject. We merely want to point out that a good branching criterion indeed is important as to the performance of the proposed algorithm.

## §6.4 Testing Results on Procedures H1 and H2

Heuristic procedure (H1) utilizes the proposed exact algorithm but limits the number of times that improvements can be made upon the incumbent. As in most heuristic approaches, we expect the procedure to produce good solutions within very reasonable computer times. Testing results on the randomly generated problems with different numbers of general constraints are reported in Table 6-11.

Table 6-11: Testing Results on Procedure H1

| # of General | CPU Time | No Improvement | | Two Improvements | |
|---|---|---|---|---|---|
| Constraints | (Optimal) | CPU time | Reach Opt. | CPU time | Reach Opt. |
| 1 | 1.52 | 1.31 | 40% | 1.45 | 100% |
| 2 | 2.18 | 1.62 | 40% | 1.95 | 80% |
| 3 | 3.02 | 2.01 | 30% | 2.77 | 90% |
| 4 | 9.10 | 3.25 | 10% | 6.04 | 60% |
| 5 | 10.64 | 4.12 | 30% | 7.02 | 70% |
| 6 | 16.02 | 6.47 | 30% | 10.91 | 70% |

The figures in Table 6–11 indicate that for many problems, an optimal solution is located after only a few (if any) improvements have been obtained. By restricting the number of possible improvements allowed, the average computer times are considerably reduced. For example, in the case that at most two improvements are allowed, optimal solutions are obtained in about 80% of the testing problems and the percentages of average time savings range from 5% to 32%. In general, the time savings become more significant when the problem becomes more difficult to solve, as demonstrated in the last three rows of Table 6–11. The quality of the solutions, as measured by their percentage deviations from the optimal values, depends upon how many improvements are requested. In the case of no improvement, the deviations of the objective values can be as high as 20%. For two improvements, the deviations are all under 5% in this set of problems. However, there is no ruie of thumb which we can provide at this point to assure the quality of the solutions, and this is a major drawback of this approach.

Many other problems were tested. The results will not be reported here since they all have about the same pattern as presented in Table 6–11. However, we notice that a) initial improvements are more significant and pronounced in terms of the objective value than those at later stages, b) the quality of the final solutions obtained are fairly decent. All the solutions are within 10% of the optimal objective values if at least two improvements are requested, and c) computer time savings tend to increase as the problem requires more computational work. Unfortunately, we are not able to draw any statistically significant conclusions from the data on hand.

For the heuristic procedure (H2), the testing was conducted on problems with a varying number of binary–valued constraints as well as with a varying number of general constraints. For each problem, we assigned two levels [illegible], namely, within 1% and 5% of the optimal objective values, respectively. The numerical results are in Tables 6–12 and 6–13.

Table 6–12: Testing Results on H2- Varying # of BVC

| # of Binary | CPU Time | Within 5% of Opt. | | Within 1% of Opt. | |
|---|---|---|---|---|---|
| Valued Const. | (Optimal) | CPU time | Reach Opt. | CPU time | Reach Opt. |
| 4 | 1.90 | 1.53 | 80% | 1.59 | 90% |
| 6 | 2.34 | 1.99 | 80% | 2.29 | 100% |
| 8 | 3.02 | 2.29 | 60% | 2.67 | 100% |
| 10 | 3.28 | 2.39 | 60% | 2.92 | 100% |
| 12 | 3.69 | 2.59 | 70% | 3.22 | 100% |
| 14 | 4.07 | 2.88 | 70% | 3.80 | 90% |
| 16 | 4.30 | 3.57 | 70% | 4.23 | 90% |

We found that the testing results are generally encouraging. As demonstrated in Table 6–12, if an accuracy of 5% is desired, optimal solutions are located in about 70% of the problems. By increasing the accuracy criterion from 5% to 1%, only 3 out of the 70 tested problems did not find an optimal solution. The computer time savings are not substantial if the 1% criterion is adopted. However, if we are satisfied with the 5% accuracy, computer time savings can be as high as 43% in some cases. As the problems become more difficult to solve in terms of the computational effort, optimal solutions are obtained less frequently, as shown in Table 6–13. It appears that for a difficult problem, there exist many potential solutions with objective values close to the optimal value. They can shield the optimal solution from being uncovered.

Table 6–13: Testing Results on H2– Varying # of Gen. Const.

| # of General Constraints | CPU Time (Optimal) | Within 5% of Opt. | | Within 1% of Opt. | |
|---|---|---|---|---|---|
| | | CPU time | Reach Opt. | CPU time | Reach Opt. |
| 1 | 1.52 | 1.43 | 100% | 1.51 | 100% |
| 2 | 2.18 | 1.84 | 40% | 2.01 | 80% |
| 3 | 3.02 | 2.29 | 60% | 2.67 | 100% |
| 4 | 9.10 | 6.00 | 50% | 7.78 | 80% |
| 5 | 10.64 | 6.02 | 30% | 8.96 | 60% |
| 6 | 16.02 | 10.48 | 70% | 14.27 | 90% |

Procedures (H1) and (H2) apply different philosophies in getting good solutions and it is not easy to compare them. Both procedures can be time–consuming if very good heuristic solutions are required. Procedure (H2) is capable of generating many good solutions because it examines all the branches, but with a weaker fathoming criterion. It is conceivable that different circumstances may warrant different heuristic procedures.

## §6.4 Testing Results on Procedure H3

In Section 5.3, we discussed the procedure (H3) in terms of the formulation $(P')$. Testing problems are still generated according to the original formulation $(P)$ since we had established a systematic way of doing this. Some modifications of the algorithm (H3) are required in order to properly accommodate the alternative formulation $(P)$, but the modifications can be easily made. Before we proceed with the testing, we need first to define several tolerance levels which are relevant to whether we consider a particular solution to be acceptable and when the algorithm should be terminated. Recall that a solution $x_c$ is considered to be acceptable if for each general constraint $a_k x \geq b_k$, $k = 2, 3, \ldots, l_1$, the inequality

$$a_k \mathbf{x}_c \geq b_k(1 - p\%)$$

holds for a given $p$. The tolerance level of 5% is assumed in our testing.

The values of three parameters are required to fully describe the stopping rules. Using the notation in Section 5.3, rule (a) is expressed as

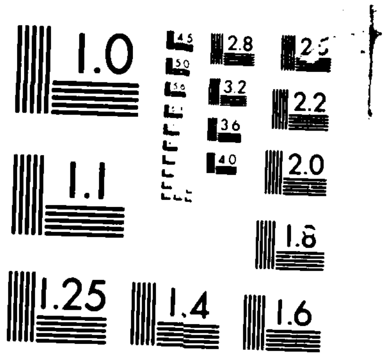
$$\underline{Z} \geq (1 - q\%) Z_D^*$$

and the task is to determine an appropriate $q$. For the testing cases, we allow a 5% gap between $\underline{Z}$ and $Z_D^*$, i.e., $q$=5. These values for $p$ and $q$ are selected simply because we think that they are of reasonable magnitude. The primary objective of the stopping criterion (b) is to assure that the iterative scheme is of finite termination. The number of iterations allowed is assumed to be 100 here. This number should provide enough room to let the algorithm fully develop. As for the criterion (c), we stop the iterative searching process when we find at least 5 acceptable solutions. When the algorithm is applied to larged sized problems, these particular values should be changed in order to assure that a reasonable number of acceptable solutions can be located.

The numerical experiment conducted on (H3) is rather limited. Extensive studies are required before we can draw any definite conclusions as to the effectiveness of this algorithm. Since the algorithm is designed to accommodate special requirements, it would not be appropriate to compare it with other heuristic approaches. The best way to evaluate the algorithm is to apply it to real problems and investigate whether good alternative proposals are generated. Unfortunately, no such problems are available for testing.

AD-A169 256 AN IMPLICIT ENUMERATION ALGORITHM WITH BINARY-VALUED 2/2
CONSTRAINTS(U) STANFORD UNIV CA SYSTEMS OPTIMIZATION
LAB Y WAN MAR 86 SOL-86-5 N00014-85-K-0343

UNCLASSIFIED F/G 9/2 NL



MICROCOP'   CHART

The problems tested are those referred to in Table 6-13 with three or more general constraints. The relaxation coefficient $W$ in Section 5.3 takes the value 1.0. Testing results are reported in Table 6-14.

**Table 6-14** : Testing Results on Procedure H3

| # of General Constraints | Acceptable Solutions Found | CPU Time (Seconds) |
|---|---|---|
| 3 | 2.1 | 1.05 |
| 4 | 2.5 | 1.32 |
| 5 | 3.9 | 3.07 |
| 6 | 4.6 | 4.11 |

As the problem becomes more difficult to solve, (H3) tends to find more acceptable solutions. This is encouraging since real problems often are complex and it certainly is helpful to have many alternative solutions. As expected, the CPU times are small since we adopted the special knapsack algorithm. The number of acceptable solutions should increase as we narrow the gap between $\underline{Z}$ and $Z_D^*$. Additional tests can be conducted along this line. The effect of $W$ on the performance of the procedure can also be addressed by the sensitivity study.

# CHAPTER 7

# CONCLUSIONS

*We summarize our findings in the first section. Directions for future studies, as well as some concluding remarks, are given in Section 7.2.*

## §7.1 Summary

We have asserted in Chapter 1 that an IP problem with special structure should call for a special-purpose IP algorithm. Throughout this dissertation, we found that this is indeed the case. By applying the specially designed MCIP/BVC algorithm, we can solve the MCIP/BVC problems much more effectively. In the extreme case where there is only one general constraint, the proposed algorithm performed much better than all other tested algorithms. However, we should emphasize that if the problems become considerably less structured, as in the case where we have many general constraints, the specially designed algorithm may be inappropriate to adopt.

In extensive testing, the performance of the MCIP/BVC algorithm generally has been good. One conjecture that we have drawn is that the efficiency of the proposed algorithm is very closely related to the ratio, $r$, between the number of general constraints and the number of BVC's. This ratio serves as an indicator to predict whether the MCIP/BVC algorithm can be effectively applied. When $r$ becomes larger, the proposed algorithm generally becomes more efficient. Other factors, such as the total number of binary variables and the number of BVC's, are less important than the ratio $r$. The rate of CPU time growth for the proposed algorithm, as demonstrated in Chapter 6, is usually slower than that for the other approaches tested. For one set of problems for which the computational results are available in the literature, the MCIP/BVC algorithm out-performed the existing algorithm by a factor of about 4.

The solving of integer programming problems is inherently time-consuming. We have introduced three heuristic procedures to reduce the amount of CPU time required. Each approach has its own special characteristics that make it suitable for certain circumstances. The general-purpose heuristic algorithms (H1) and (H2) are important because any enumeration type algorithm can readily be generalized in the ways described in Chapter 5 in order to operate as a heuristic algorithm. The computational results are encouraging. The heuristic procedure (H3) is very useful because it highlights certain concerns that sometimes are encountered by a decision-maker. The importance of the approach derives not only from the way in which we deal with the problem but also from the philosophy behind the development of the algorithm. We feel that both the problems with special structure and the problems with specific requirements merit the development of a special-purpose algorithm. At times, the two concerns can be utilized simultaneously.

## §7.2 Future Study

We mentioned in Chapter 6 that the current implementation of the proposed algorithm is not very efficient. In order to fully understand the scope of the approach and compare results with other algorithms, we need to improve the coding and conduct more numerical experiments. Both the exact algorithm and the heuristic algorithms should be applied on large-sized problems to form a complete spectrum of computational results.

This dissertation deals with binary integer programming problems. However, many real-world problems are more complicated and involve general integer variables and/or continuous variables. It would be interesting to know whether the proposed algorithm can serve as a basis to solve these mixed type integer programming problems. Furthermore, generalizing the heuristic approaches, especially the procedure (H3), to accommodate these non-binary variables is also important and worth exploring.

In Chapter 3, we introduced four types of group contingent constraints. Two of them are binary-valued, and it is MCIP problems containing these types of constraints that the proposed algorithm is designed to solve. The other two are not binary-valued constraints but often appear in real-world problems. Special algorithms should be developed to take advantage of these special structures as well.

For heuristic procedure (H3), it would be desirable to use real-world problems to test its effectiveness. We are interested to know whether the procedure can really aid the decision-makers and what improvements can be made. It is important to realize that from a practical point of view, a set of good feasible solutions may be more valuable than a single optimal solution. The procedure (H3) should serve as an example to illustrate this point.

# APPENDIX

## The NBA Scheduling Problem

### §A.1 Formulation

The National Basketball Association (NBA) scheduling problem is a MCIP with binary–valued constraints. It resembles the classical assignment problem in many aspects. A major difference between the two problems is that an optimal basic feasible solution of the LP–relaxation of the NBA scheduling problem is not necessarily integer valued. This property makes the problem an ideal candidate for testing the MCIP/BVC algorithm.

Suppose that we have $n$ basketball teams, each based in a different city. Among the $n$ teams, $[\frac{n}{2}]$ teams will play at home and $[\frac{n}{2}]$ teams will play away from home, where $[k]$ denotes the largest integer which is less than or equal to $k$. If $n$ is an even number, every team is required to play. If $n$ is odd, one team will not play. For practical purposes and the ease of presentation, we assume that $n$ is even. The objective is to find a playing schedule which minimizes the total traveling distance.

Denote the binary variables by

$$x_{ij} = \begin{cases} 1, & \text{if team } i \text{ plays at city } j, \\ 0, & \text{otherwise,} \end{cases}$$

where $i = 1, \ldots, n$ and $j = 1, \ldots, n$. $x_{ii} = 1$ implies that team $i$ is playing at home. The SOS constraints,

$$\sum_{j=1}^{n} x_{ij} = 1, \quad i = 1, \ldots, n,$$

state that every team has to play and can play only at one city. If team $j$ stays at home, at most one team can visit city $j$. The constraints,

$$x_{jj} - \sum_{i \neq j} x_{ij} \geq 0, \quad j = 1, \ldots, n, \tag{1}$$

represent this requirement. When $n$ is an even number, the requirement should read that if team $j$ stays at home, exactly one team will visit city $j$. The *greater than or equal to* sign in (1) can be replaced by an *equal* sign. However, we maintain the inequality relationship in (1) to highlight the contingency property. In addition, we shall define the coefficients of the objective function in such a way that one other team definitely will visit city $j$ when team $j$ plays at home.

The traveling distance of team $i$ depends on its current location, which may be either its home city or the city in which it played its last game. Let

$$c_{ij} = \text{distance that team } i \text{ needs to travel to reach city } j,$$

where $i = 1, \ldots, n$ and $j = 1, \ldots, n$. If team $i$ is not allowed to play at city $j$, simply set $c_{ij} = \infty$. The NBA scheduling problem takes the form

$$\begin{array}{lll} \text{Minimize} & \sum_{i=1}^{n} \sum_{j=1}^{n} c_{ij} x_{ij} & \\ \text{subject to} & x_{jj} - \sum_{i \neq j} x_{ij} \geq 0, \qquad j = 1, 2, \ldots, n & \\ & \sum_{j=1}^{n} x_{ij} = 1, \qquad i = 1, 2, \ldots, n & (S) \\ & x_{ij} \in \{0, 1\} \quad \forall\, i, j. & \end{array}$$

To guarantee that one other team definitely will visit city $j$ when team $j$ is at home, we set $c_{ii} = M$, a very large positive number that is larger than any of the travel distances, for all $i$. Setting $c_{ii}$ this way assures that exactly $\lceil \frac{n}{2} \rceil$ teams play at home, as we will see in Section A.2. When team $j$ is not allowed to play at home, set $c_{jj} = \infty$ as before.

## §A.2 Properties and Testing Problems

We shall first show that exactly $\frac{n}{2}$ teams will play at home. Recall that $n$ is assumed to be an even number and the $x_{ij}$ are binary variables. Summing up all contingent constraints and SOS constraints, we have

$$2\sum_{j=1}^{n} x_{jj} \geq n \quad \Longrightarrow \quad \sum_{j=1}^{n} x_{jj} \geq \frac{n}{2}.$$

Since the $c_{jj}$ are very large positive numbers and the problem is to be minimized, as many of the $x_{jj}$ as possible should take the value zero. This implies that

$$\sum_{j=1}^{n} x_{jj} = \frac{n}{2},$$

which states that exactly $\frac{n}{2}$ teams play at home. By the contingent constraints, the remaining $\frac{n}{2}$ teams can not visit city $j$ when $x_{jj} = 0$ and only one team based in another city can visit city $j$ when $x_{jj} = 1$.

If we selected the $\frac{n}{2}$ teams, e.g., team 1 through team $\frac{n}{2}$, to play at home, then the problem $(S)$ can be simplified substantially. After eliminating the variables which are forced to take value zero, the problem $(S)$ becomes

$$\begin{array}{ll} \text{Minimize} & \sum_{i=1}^{\frac{n}{2}} c_{ii} \;+\; \sum_{i=\frac{n}{2}+1}^{n} \sum_{j=1}^{\frac{n}{2}} c_{ij} x_{ij} \\ \text{subject to} & 1 - \sum_{i=\frac{n}{2}+1}^{n} x_{ij} \geq 0, \qquad j = 1, \ldots, \frac{n}{2} \\ & \sum_{j=1}^{\frac{n}{2}} x_{ij} = 1, \qquad i = \frac{n}{2}+1, \ldots, n \\ & x_{ij} \in \{0, 1\} \quad \forall\, i, j. \end{array} \qquad (S')$$

Note that the greater than or equal to sign in the contingent constraints can be replaced by the equal sign as discussed early. The new problem $(S')$ then is the classical assignment problem. Solving $(S')$ is equivalent to solving its LP-relaxation.

When all the teams currently are at home, e.g., just before the beginning of the season, $(S)$ takes a special form. The coefficient $c_{ij}$ should equal $c_{ji}$ for all $j \neq i$.

This makes the coefficient matrix $C = \{c_{ij}\}$ a symmetric matrix. Furthermore, for any playing schedule, its reverse playing schedule, defined by interchanging the home teams and the visiting teams, is equally good. For the testing problems, we assume that the matrix $C$ is symmetric, which eliminates the necessity of specifying the current location of each team when generating the testing problems.

The testing problems were generated by Bean (1980) in a straightforward fashion. He first selected 20 U.S. major cities to form the reference group. Each set of problems had a fixed (even) value of $n < 20$, depending upon how many teams are needed. For each problem in the set, he randomly selected the required number of cities from the reference group. The coefficient $c_{ij}$ is assigned to be the actual distance between cities $i$ and $j$. All other constraints are explicitly defined and they are not random in nature.

# BIBLIOGRAPHY

1. Abadie, J. ed. (1970). *Integer and Nonlinear Programming*. American Elsevier.

2. Armstrong, R.D., W.D. Cook, and D. Kung (1981). The Multiyear Capital Budgeting Problem: A Lagrangean Relaxation Approach. *Res. Rep.* CCS **394**, Center for Cybernetic Studies, U. of Texas at Austin.

3. Balas, E. (1965). An Additive Algorithm for Solving Linear Programs with Zero-One Variables. *Oper. Res.* **13**, 517–546.

4. Balas E. and C.H. Martin (1978). Pivot and Complement – A Heuristic for 0-1 Programming. *Manage. Sci. Res. Rep.* No. **414**, Carnegie–Mellon University.

5. Balas E. and R.G. Jeroslow (1975). Strengthening Cuts for Mixed Integer Programs. *Manage. Sci. Res. Rep.* No. **359**, Carnegie–Mellon University.

6. Balinski, M.L. and R.E. Quandt (1964). On an Integer Program for a Delivery Problem. *Oper. Res.* **12**, 300–304.

7. Balinski, M.L. (1965). Integer Programming: Methods, Uses and Computation. *Manage. Sci.* **12**, 253–313.

8. Beale, E.M.L. (1965). Survey of Integer Programming. *Oper. Res. Quarterly* **16**, 219–228.

9. Beale, E.M.L. (1979). Branch and Bound Methods for Mathematical Programming Systems. In Hammer, Johnson, and Korte.

10. Beale, E.M.L. and J.A. Tomlin (1969). Special Facilities in a General Mathematical Programming System for Non-convex Problems Using Ordered Sets of Variables. 8.21–8.28 in Lawrence.

11. Bean, J.C. (1980). An Additive Algorithm for the Multiple Choice Integer Program. Ph.D. *Dissertation* , Depart. of Oper. Res., Stanford University.

12. Bellman, R.E. and M. Hall, jr. eds. (1960). Combinatorial Analysis. *Proc. Tenth Symp. in Appl. Math. of the Am. Math. Soc.*.

13. Benichou, M., J.M. Gauthier, P. Girodet, G. Hentges, G. Ribière, and O. Vincent (1971). Experiments in Mixed Integer Linear Programming. *Math. Prog.* **1**, 76–94.

14. Christofides, N., A. Mingozzi, P. Toth, and C. Sandi eds. (1979). *Combinatorial Optimization* . John Wiley and Sons.

15. Crowder, P., M. Held, and P. Wolfe (1974). Validation of Subgradient Optimization. *Math. Prog.* **6**, 62–88.

16. Dantzig, G.B., D.R. Fulkerson, and S.M. Johnson (1954). Solution of a Large Scale Travelling Salesman Problem. *Oper. Res.* **2**, 393–410.

17. Dantzig, G.B. (1963). *Linear Programming and Extensions.* Princeton University Press.

18. Dantzig, G.B. and R.M. Van Slyke (1967). Generalized Upper Bounding Techniques. *J. Computer Syetem Science* **1**, 213–226.

19. Davis, R.E., D.A. Kendrick, and M. Weitzman (1971). A Branch-and-Bound Algorithm for 0-1 Mixed Integer Programming Problems. *Oper. Res.* **19**, 1036–1044.

20. Driebeek, N.J. (1966). An Algorithm for the Solution of Mixed Integer Programming Problems. *Manage. Sci.* **12**, 576–587.

21. Garey, M. and D. Johnson (1978). *Computer and Intractability: A Guide to the Theory of NP-Completeness.* San Francisco: Freeman and Co..

22. Garfinkel, R.S. (1979). Branch and Bound Methods for Integer Programming. In Christofides, Mingozzi, Toth, and Sandi.

23. Garfinkel, R.S. and G.L. Nemhauser (1972). *Integer Programming.* John Wiley and Sons.

24. Gauthier, J.M. and G. Ribière (1977). Experiments in Mixed Integer Linear Programming Using Pseudo-Costs. *Math. Prog.* **12**, 26–47.

25. Geoffrion, A.M. (1969). An Improved Implicit Enumeration Approach for Integer Programming. *Oper. Res.* **17**, 437–454.

26. Geoffrion, A.M. and R.E. Marsten (1972). Integer Programming Algorithms: A Framework and State-of-the-Art Survey. *Manage. Sci.* **18**, 465–491.

27. Geoffrion, A.M. (1974). Lagrangian Relaxation for Integer Programming. *Math. Prog. Study* **2**, 82–114.

28. Geoffrion, A.M. (1976). Guided Tour of Recent Practical Advances in Integer Linear Programming. *Omega-International Jour. of Manage. Sci.* **4(1)**, 49-57.

29. Glover, F. (1965). A Multiphase-Dual Algorithm for the Zero-One Integer Programming Problem. *Oper. Res.* **13**, 879–919.

30. Glover, F. (1968). Surrogate Constraints. *Oper. Res.* **16**, 741–749.

31. Glover, F. (1969). Integer Programming over a Finite Additive Group. *SIAM Jour. of Control* **7**, 213–231.

32. Gomory, R.E. (1958). Outline of an Algorithm for Integer Solutions to Linear Programs. *Bull. Amer. Math. Soc.* **64**, 275–278.

33. Gomory, R.E. (1960). Solving Linear Programming Problems in Integers. pp. 211–216 in Bellman and Hall.

34. Gomory, R.E. (1963). An Algorithm for Integer Solutions to Linear Programs. pp. 269–302 in Graves and Wolfe.

35. Gomory, R.E. (1965). On the Relation between Integer and Noninteger Solutions to Linear Programs. *Proc. of the National Academy of Science* **53**, 260–265.

36. Gorry, G.A. and J.F. Shapiro (1971). An Adaptive Group Theoretic Algorithm for Integer Programming Problems. *Manage. Sci.* **17**, 285–306.

37. Graves, R.L. and P. Wolfe eds. (1963). *Recent Advances in Mathematical Programming.* McGraw–Hill.

38. Hammer, P.L., E.L. Johnson, and B.H. Korte eds. (1979). Discrete Optimization II. *Annals of Discrete Mathematics* **5**, North-Holland.

39. Hanssmann F. (1968). *Operations Research Techniques for Capital Investment.* John Wiley & Sons.

40. Healy, W.C. (1964). Multiple Choice Programming. *Oper. Res.* **12**, 122–138.

41. Hillier, F.S. (1969a). Efficient Heuristic Procedures for Integer Linear Programming with an Interior. *Oper. Res.* **17**, 600–637.

42. Hillier, F.S. (1969b). A Bound-and-Scan Algorithm for Pure Integer Linear Programming with General Variables. *Oper. Res.* **17**, 638–679.

43. Hillier, F.S. and G.J. Liberman (1980). *Operations Research.* Holden–Day.

44. Holzman, A.G. ed. (1979). *Operations Research Support Methodology.* Dekker.

45. Jeroslow, R.G. (1979). The Theory of Cutting-Plane. In Christofides, Mingozzi, Toth, and Sandi.

46. Johnson, E.L. (1979). On the Group Problem and a Subadditive Approach to Integer Programming. pp. 97-112 in Hammer, Johnson, and Korte.

47. Johnson, E.L. (1981). On the Generality of Subadditive Characterization of Facets. *Math. of Oper. Res.* **6**, 101-112.

48. Kaul, R.N. (1965). An Extension of Generalized Upper Bounding Techniques for Linear Programming. *Rep. ORC 65-27*, Operations Research Center, Univ. of California at Berkeley.

49. Kochman, G.A. (1976). Decomposition in Integer Programming. *Technical Report* No. **76-21**, Depart. of Oper. Res., Stanford University.

50. Lasdon, L.S. and R.C. Terjung (1971). An Efficient Algorithm for Multi-Item Scheduling. *Oper. Res.* **19**, 998–1022.

51. Lawler, E.L. and D.E. Wood (1966). Branch-and-Bound Methods: A Survey. *Oper. Res.* **14**, 699–719.

52. Lawrence, J. ed. (1969). *Proc. Fifth Int. Conf. of Operational Research.* Venice, Tavestock Pub., London.

53. Lemke, C.E. and Speilberg, K. (1967). Direct Search Algorithm for Zero-One and Mixed Integer Programming. *Jour. Oper. Res. Soc. Amer.* **15(5)**.

54. Little, John D.C., K.G. Murty, D.W. Sweeney, and C. Kavel (1963). An Algorithm for the Traveling Salesman Problem. *Oper. Res.* **11**, 972–989.

55. Lorie, J. and L.J. Savage (1955). Three Problems in Capital Rationing. *Jour. of Bus.* **28**, 229–239.

56. Markowitz, H.M. and A.S. Manne (1957). On the Solution of Discrete Programming Problems. *Econometrica* **25**, 84–110.

57. Martin, R.K. (1980). The Optimization of Integer Programs with Special Ordered Sets of Variables. Ph.D. *Dissertation*, Univ. of Cincinnati.

58. Mevert, P and U. Suhl (1977). Implicit Enumeration with Generalized Upper Bounds. *Annals of Discrete Mathematics* **1**, 392–402.

59. Mitten, L.G. (1970). Branch–and–Bound Methods: General Formulation and Properties. *Oper. Res.* **18**, 24–34.

60. Nauss, R.M. (1975). The 0–1 Knapsack Problem with Multiple Choice Constraints. *Working Paper*, U. Missouri–St. Louis, March 1975, Revised May 1976.

61. Reardon, K.J. (1974). Decomposition of Dual Angular Integer Programs with Applications to Multi-Stage Capital Budgeting. Ph.D. *Dissertation* , Depart. of Oper. Res., Stanford University.

62. Savage, J.E. (1976). *The Complexity of Computing* . John Wiley and Sons.

63. Senju, S. and Y. Toyoda (1968). An Approach to Linear Programming with 0-1 Variables. *Manage. Sci.* **15**, B196–B207.

64. Shapiro, J.F. (1968a). Dynamic Programming Algorithms for the Integer Programming Problem–I: The Integer Programming Problem viewed as a Knapsack Problem. *Oper. Res.* **16**, 103–121.

65. Shapiro, J.F. (1968b). Group Theoretic Algorithms for the Integer Programming Problem II: Extension to a General Algorithm. *Oper. Res.* **16**, 928–947.

66. Shapiro, J.F. (1970). Turnpike Theorems for Integer Programming Problems. *Oper. Res.* **18**, 432–440.

67. Sinha, P. and A. Zoltners (1979a). The Multiple Choice Knapsack Problem. *Oper. Res.* **27**, 503-515.

68. Sinha, P. and A. Zoltners (1979b). A Multiple–Choice Integer Programming Algorithm. Graduate School of Management, Northwestern University.

69. Tomlin, J.A. (1970). Branch and Bound Methods for Integer and Non-Convex Programming. pp. 437–450 in Abadie.

70. Toregas, C., R. Swain, C. Revelle, and L. Bergman (1971). The Location of Emergency Service Facilities. *Oper. Res.* **19**, 1363–1373.

71. Toyoda, Y. (1975). A Simplified Algorithm for Obtaining Approximate Solutions for Zero-One Programming Problems. *Manage. Sci.* **21**, 1417–1427.

72. Trauth, C.A. and R.E. Woolsey (1968). MESA; A Heuristic Integer Linear Programming Technique. *Res. Rep. SC-RR-68-299* , Sandia Labs., Albuquerque, New Mexico.

73. Weingartner, H.M. (1963). *Mathematical Programming and the Analysis of Capital Budgeting Problems.* Prentics-Hall.

74. Weingartner, H.M. (1966). Capital Budgeting of Interrelated Projects: Survey and Synthesis. *Manage. Sci.* **12**, 485–516.

75. Wosley, L.A. (1969). Mixed Integer Programming: Discretization and the Group Theoretic Approach. *Technical Report* No. **42**, Operations Research Center, MIT.

76. Wolsey, L.A. (1979). Cutting Plane Methods. pp. 441-466, in Holzman.

77. Wosley, L.A. (1980). Heuristic Analysis, Linear Programming and Branch–and–Bound. *Math. Prog. Study* **13**, 121–134.

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

| | | |
|---|---|---|
| **REPORT DOCUMENTATION PAGE** | | READ INSTRUCTIONS BEFORE COMPLETING FORM |
| 1. REPORT NUMBER<br>SOL 86-5 | 2. GOVT ACCESSION NO. | 3. RECIPIENT'S CATALOG NUMBER |
| 4. TITLE (and Subtitle)<br>An Implicit Enumeration Algorithm with Binary-Valued Constraints | | 5. TYPE OF REPORT & PERIOD COVERED<br>Technical Report |
| | | 6. PERFORMING ORG. REPORT NUMBER |
| 7. AUTHOR(s)<br>Yieh-Hwang Wan | | 8. CONTRACT OR GRANT NUMBER(s)<br>N00014-85-K-0343 |
| 9. PERFORMING ORGANIZATION NAME AND ADDRESS<br>Department of Operations Research - SOL<br>Stanford University<br>Stanford, CA 94305 | | 10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS<br>NR-047-064 |
| 11. CONTROLLING OFFICE NAME AND ADDRESS<br>Office of Naval Research - Dept. of the Navy<br>800 N. Quincy Street<br>Arlington, VA 22217 | | 12. REPORT DATE<br>March 1986 |
| | | 13. NUMBER OF PAGES<br>107 pp. |
| 14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office) | | 15. SECURITY CLASS. (of this report)<br>UNCLASSIFIED |
| | | 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE |

16. DISTRIBUTION STATEMENT (of this Report)

This document has been approved for public release and sale; its distribution is unlimited.

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

| | |
|---|---|
| contingent constraints | multiple choice constraints |
| capital budgeting model | branch-and-bound |
| integer programming | |

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

Please see next page.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

END

DTIC

7-86